(4)

BBN Laboratories Incorporated

A Subsidiary of Bolt Beranek and Newman Inc.

Report No. 6813

INTEGRATION OF SPEECH AND NATURAL LANGUAGE

INTERIM REPORT

D. Ayuso, Y. Chow, A. Haas, R. Ingria, S. Roucos, R. Scha, D. Stallard

April 1988

Submitted to:

Advanced Research Projects Agency
1400 Wilson Blvd.
Arlington, VA 22209

Office of Naval Research
Department of the Navy
Arlington, VA 22217-5000

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER BBN Report No. 6813	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Integration of Speech and Natural Language Interim Report		5. TYPE OF REPORT & PERIOD COVERED Interim Report Dec 30, 1986 - March 31, 1988 6. PERFORMING ORG. REPORT NUMBER BBN Report No. 6813
7. AUTHOR(s) D. Ayuso, Y. Chow, A. Haas, R. Ingria, S. Roucos, R. Scha, D. Stallard		8. CONTRACT OR GRANT NUMBER(s) N00014-87-C-0085
9. PERFORMING ORGANIZATION NAME AND ADDRESS BBN Laboratories 10 Moulton Street Cambridge, MA 02238		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Office of Naval Research Department of the Navy Arlington, Virginia 22217-5000		12. REPORT DATE April 1988 13. NUMBER OF PAGES 84
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) Unclassified 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)
Distribution of the document is unlimited. It may be released to the Clearinghouse, Dept. of Commerce, for sale to the general public.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)
Unification grammars, parsing, natural language processing, Compositional semantics, Intensional logic, Higher order logic speech recognition, speech understanding, hidden Markov models.

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)
We present in this interim report our work on integrating speech and natural language processing for speech understanding. We describe the components of the system: the unification grammar and corresponding parser, the higher order intensional logic and the type system used for semantic interpretation, and the search stracegy used for speech understanding.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

Report No. 6813

ARPA Order Number 5947
Contract Number N00014-87-C-0085
Contract Duration: 30 Dec 1986 - 29 Dec 1988
Principal Investigators: Dr. Salim Roucos (617)873-3452
Dr. Remko Scha (617)873-2670

INTEGRATION OF SPEECH AND NATURAL LANGUAGE

INTERIM REPORT

D. Ayuso, Y. Chow, A. Haas, R. Ingria, S. Roucos, R. Scha, D. Stallard

April 1988

Table of Contents

Executive Summary 1

1. The Syntactic Component 5

1.1 The Grammar Formalism 5
1.1.1 The Relation between The Grammar and the Lexicon 6
1.1.2 Optional Elements 8
1.1.3 "Meta-rules" and Feature Value Default Mechanisms 8
1.1.4 Trace Flags 8
1.2 The Parsing Algorithm 9
1.3 Qualitative Measures of Coverage 11
1.3.1 The Top Level Constructions 12
1.3.2 Clausal Constructions 12
1.3.3 Verb Phrase Constructions 14
1.3.4 Auxiliary Constructions 15
1.3.5 Noun Phrases 16
1.3.6 Adjective Phrase Constructions 19
1.3.7 Adverbial Constructions 20
1.3.8 Other Constructions 20
1.4 Quantitative Measures of Coverage 21
1.4.1 Grammar Size 21
1.4.2 Syntactic Coverage 21
1.4.3 Perplexity 21
1.4.4 Ambiguity 23
1.4.5 Overgeneration 25
1.5 Future Plans 26
1.5.1 Extending Coverage 26
1.5.2 Reducing Spurious Ambiguity 28
1.5.3 Changes to the Grammar Formalism 28

2. The Semantic Component 29

2.1 Introduction 29
2.2 The Nature of Semantic Knowledge 29
2.3 System Design Overview 33
2.4 The Logic 35
2.5 Example of Processing 38
2.6 The Semantic Framework System 40
2.6.1 Introduction 40
2.6.2 Logical Expressions as Data Abstractions 41
2.6.3 Functions for Defining Constants 42
2.6.4 Functions for Extending the Language 42
2.6.5 Translations and Transformations 43
2.6.6 Functions for Comparing Types 44
2.6.7 Syntax Checkers for Logic Expressions 46

2.7 Accomplishments over the Last Year 46
2.7.1 Implementation Status 46
2.7.2 Theoretical Issues and Publications 47
2.7.3 Future Work 47

3. Speech and Natural Language Integration 49

3.1 Speech 50
3.2 Integration of Speech and Syntax 52
3.2.1 The Time-Synchronous Speech Parser 52
3.2.2 The Word-Synchronous Speech Parser 54
3.3 Integrating Semantics 56
3.4 System Implementation 57
3.4.1 Silence Handling 57
3.4.2 Search Strategies 57
3.5 Current Status and Future Work 59

References 61

APPENDIX A. A Parsing Algorithm for Unification Grammar 65
Andrew Haas

A.1 Basic Concepts 67
A.2 Operations on Sets of Rules and Terms 68
A.3 The Parser without Empty Symbols 70
A.4 The Parser with Empty Symbols 73
A.5 The Parser with Top-Down Filtering 75
A.6 Discussion and Implementation Notes 81

List of Figures

Figure 1: Architecture of The Natural Language Processing System 2
Figure 1-1: BBN ACFG Parsing Algorithm 11
Figure 1-2: BBN ACFG Resource Management Training and Test Corpus Coverage 22
Figure 2-1: Screen Display of Parse Tree 39
Figure 3-1: Dynamic Time Warping (DTW) algorithm 1 51
Figure 3-2: Dynamic Time Warping (DTW) algorithm 2 51
Figure 3-3: Time-synchronous Lattice Parsing Algorithm 53
Figure 3-4: Word-synchronous Parsing Algorithm 55

Executive Summary

This report describes the progress during the first year of the project from January 1, 1987 to December 31, 1987. During this first year, we have focused on two major activities:

- Development of the syntax and semantics components for natural language processing.
- Integration of the developed syntax and semantics with speech for speech understanding.

To measure the coverage of the syntactic and semantic components and the performance of the integrated system, we use the DARPA 1000-word Resource Management Domain Corpus. This corpus has been used for developing a standard speech database for evaluating the performance of speech recognition algorithms developed under the Strategic Computing Program.

Our work on natural language processing included the developement of a grammar (syntax) that uses the Unification grammar formalism (an augmented context free formalism). The Unification grammar formalism, the rules, and the parsing algorithm are described in more detail in Chapter 1. The syntactic phenomenina that the grammar handles are also described. Currently, the grammar cover 85% of a training corpus and 64% of a test corpus. The training corpus is examined by the syntax developers and is used to determine the phenomena that should be handled. The test set is never examined by the syntax developers and its purpose is to allow us to estimate performance on an independent set that would be representative of the ultimate system's performance in the field.

The parsing algorithm extends the algorithm of Graham, Harrison, and Ruzzo [9] from the context-free case to incorporate unification. The new algorithm is reviewed briefly in Section 1.2 of this report and the paper by Haas [11] describing the algorithm in detail is included as Appendix A.

The semantic component uses a principled mathematical logic approach for developing a representation of meaning and the associated interpretation algorithms. Higher-order intensional logic is used as the logical language. The semantic interpreter uses 4 translation steps to derive the meaning of a parse tree of a sentence; the relationship among these levels is shown in Figure 1. First, the parse tree is converted to an expression of EFL (English-oriented Formal Language); at this level, each word (including words with multiple senses) has one EFL constant. Second, a possibly ambiguous EFL expression is translated to several expressions of WML (World Model Language). Note that each EFL constant that corresponds to an ambiguous word may have multiple WML constants, which can result in a combinatorial explosion of WML expressions derived from an EFL expresssion. Not all combinations are meaningful, however. Meaningless combinations are filtered out; they are recognized by a module that uses the *type system* of the logical language. The third translation converts the remaining set of WML expressions to expressions of DBL (Data Base Language), which are finally converted to value expressions by evaluating the DBL expression against the data base. At the value stage, presupposition failure can be detected.

We have completed the basic framework of the semantic component, implemented the type system, and have included semantic knowledge to demonstrate interpretation on a small set of sentences. In the coming year, we plan to increase the semantic coverage significantly to yield a useful system.

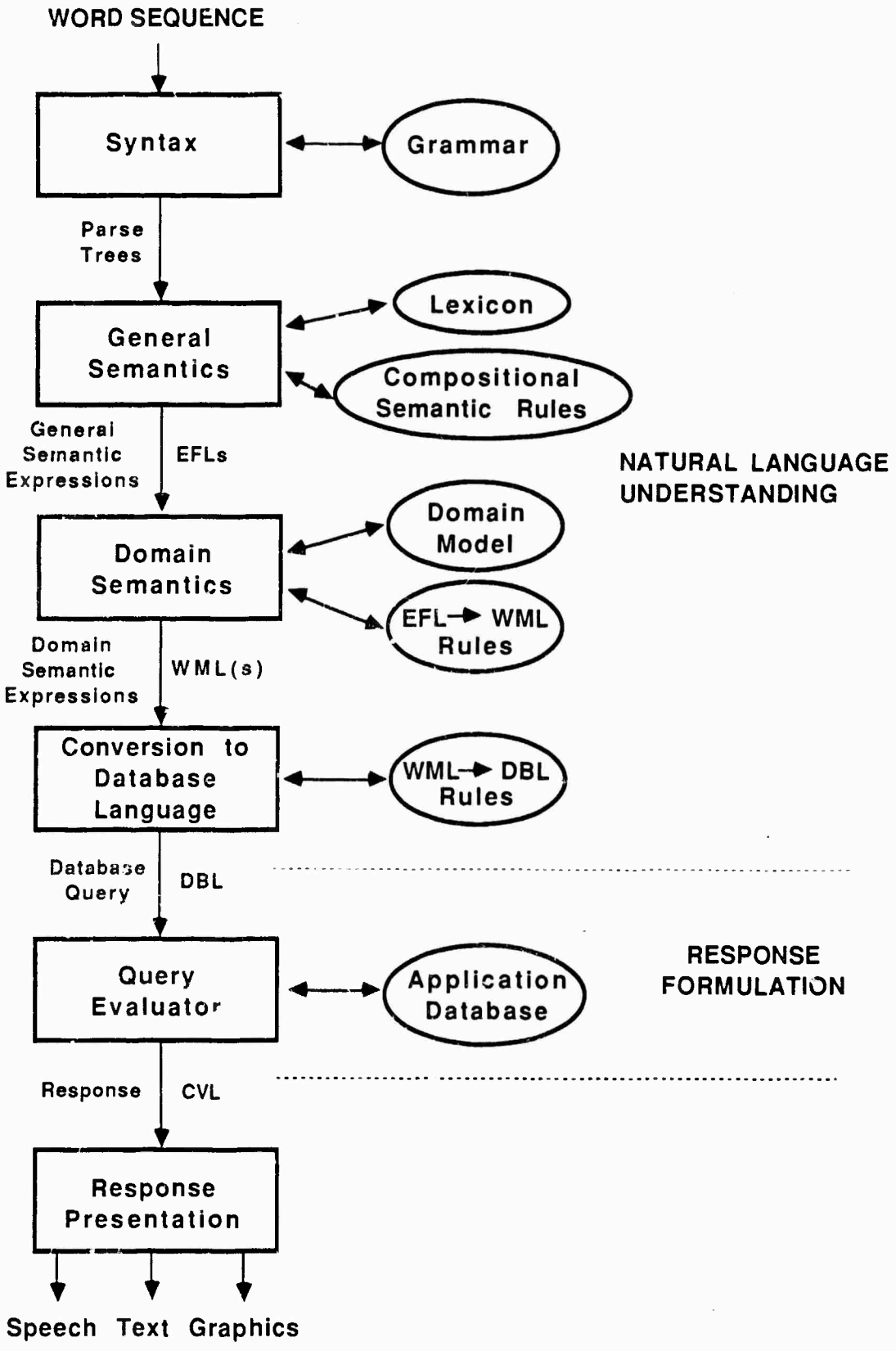


Figure 1: Architecture of The Natural Language Processing System

The work on integration has led to the development and implementation of a search strategy that integrates the unification syntax, compositional semantics, and hidden Markov word models into an algorithm that finds the best interpretation of the input speech. The search therefore integrates natural language knowledge sources and uses their constraints to find the word transcription and meaning of input speech. The search is based on the parsing algorithm that has been developed for the syntax, and applies semantics as a post process. The parser is used to find a set of grammatical sentences that have a high acoustic likelihood score given the speech. This set of sentences is ordered by decreasing likelihood. The semantic component is then applied as a post process on this set of sentences to determine the highest scoring meaningful utterance. That sentence is the recognized sentence. The search is currently implemented and has run on a few sentences. We expect in the coming year to improve the efficiency of the search, and to evaluate the speech understanding performance on a set of speakers.

In addition to the above accomplishments during this first year, we have also demonstrated the serial connection of speech recognition with a natural language component. In this case, the speech uses a language model that is different from the syntax and semantics components of natural language processing. This approach is not optimal and we think it is only applicable for applications of very low perplexity (less than 50). The integrated approach described above is required for larger perplexity tasks. Nevertheless, the serial demo is useful to demonstrate speech understanding in an actual task, due to its speed in recognition. The speech component had a vocabulary of 600 words and a finite state grammar with a perplexity of 40. The recognized word string was passed to the natural language component which interpreted the request, accessed the database system, and presented the output using the a simulation of the OSGP graphics system. This demonstration was capable of handling the demo scenario and was not considered as a robust system. The serial connection was demonstrated on two occasions: once to the program manager and once to all participants of the Strategic Computing Speech Program meeting on October 13-15, 1987.

The remaining chapters of the report describe our work on syntax in Chapter 1, on semantics in Chapter 2, and on the integration strategy in Chapter 3.

1. The Syntactic Component

This chapter describes the syntactic component of the BBN Spoken Language System. This component uses a broad-coverage grammar written in an augmented phrase structure grammar formalism and parsed using an algorithm based on the CKY algorithm for context-free grammars. Section 1.1 discusses the format of the syntactic grammar at a relatively high level. Section 1.2 introduces the algorithm that is used to parse the grammar. Sections 1.3 and 1.4 describe the coverage of the grammar, both in descriptive linguistic terms and in more quantitative measures. Section 1.5 outlines the work on the syntactic component that is currently planned, on the basis of what has already been done.

1.1 The Grammar Formalism

The BBN Spoken Language System uses a grammar formalism based on annotated phrase structure rules; this formalism is called the BBN ACFG (for Annotated Context Free Grammar). It is, therefore, in the tradition of augmented phrase structure grammars such as those of Harman [12] and Heidorn [13], [14], although its immediate inspiration is Generalized Phrase Structure Grammar (GPSG) [8]. In such grammars, rules are made up of elements that are not atomic categories but, rather, are complex symbols consisting of a category label and feature specifications. Rules in the BBN ACFG consist of grammatical symbols—e.g. representing a part of speech, such as N, for noun—that take a specified set of arguments (also referred to as *features*). These, in turn, may take arguments of their own. For example, in the current grammar, nouns and verbs contain **AGREEMENT** as an argument. **AGREEMENT**, in turn, takes the arguments **PERSON** and **NUMBER**. Arguments such as **PERSON** and **NUMBER** that do not take arguments but only assume simple feature values can have either constants or variables as values. Variables begin with a colon; constants are unary lists. For example, **PERSON** can take on one of the values **(1ST)**, **(2ND)**, **(3RD)**, and **:P**; **NUMBER**, the values **(SINGULAR)**, **(PLURAL)** and **:N**. Arguments that take arguments of their own, such as **AGREEMENT**, can also take either constants or variables as their values. Again, variables begin with a colon; constants, however, are multiple element lists whose first element is fixed across all values. For **AGREEMENT**, for example, this is **AGR**. Moreover, since the arguments to a feature such as **AGREEMENT** may themselves be either constants or variables, it is possible to have partially specified values for such features. Here are some examples of fully and partially specified arguments for **AGREEMENT**:

```
(AGR (1ST) (PLURAL))
(AGR (3RD) (SINGULAR))
(AGR (2ND) :N)       ;; (number unspecified)
(AGR :P (PLURAL))    ;; (person unspecified)
:AGR                 ;; (agreement completely unspecified)
```

Variables can be used in different elements of a rule as a means of imposing feature agreement. The following rule from the current grammar illustrates this agreement mechanism:[1]

[1]This is a simplified version of the rule; features that are irrelevant for the purposes of the present discussion have been omitted.

```
;; basic toplevel declarative clause rule, ensuring subject-verb agreement
((S ... :MOOD (WH-) ...)
 (NP :AGR :NPTYPE ...)
 (VP :AGR :NPTYPE :MOOD ...)
 (OPTSADJUNCT ...))[2]
```

This rule states that a declarative (**(WH-)**)[3] S (sentence) consists of an NP (noun phrase) followed by a VP (verb phrase) and an optional adjunct. The use of the **:AGR** variable in the NP and VP elements of the rule enforces agreement of the NP and VP in person and number. Similarly, the **:NPTYPE** variable requires that the subject NP be of the type selected by the VP (ultimately, by the head verb of the VP). Finally, the **:MOOD** variable in the S and VP elements requires that they have the same mood.

The BBN ACFG grammar is strongly typed. Each grammatical symbol has a fixed number of arguments in a fixed order. Each argument, in turn, has a fixed set of permissible values.

1.1.1 The Relation between The Grammar and the Lexicon

In phrase-structure based formalisms, there is no formally separate lexicon; lexical items are introduced by phrase structure rules just as syntactic categories (''non-terminals'') are. For example, in order for a grammar written in the ACFG formalism to contain the word ''given'', there would need to be a rule of the following sort.

```
((V (DITRANSITIVE :PASSIVE) :P :N (EDPARTICIPLE))
 (given))
```

This rule states that ''given'' is a past participle (**(EDPARTICIPLE)**), unspecified for person and number agreement (**:P :N**), that it takes a ditransitive complement structure (**(DITRANSITIVE)**), and that it may appear either in active or passive constructions (**:PASSIVE**). Note that this rule introduces ''given'' in only one of its uses, the ditransitive (as in ''We have given John the book''; ''John was given a book''). There need to be analogous rules for its other uses, as well. While it might be possible to store all these rules, the storage requirements for doing so are prohibitive. The number of rules for each lexical item is equal to the number of inflected forms of the item—singular and plural forms for nouns; positive, comparative, and superlative forms for adjectives; and all the past, present, and participial forms for verbs—multiplied by the number of subcategorization frames[4] that the lexical item may appear in. Even for a small lexicon, this will result in a large number of rules; for example, for the 1000-word Resource Management corpus, there would be over 1200 such rules for nouns, over 500 for adjectives, and slightly less than 1000 for verbs a total of approximately 2700 rules. For the multi-thousand word lexicons needed for robust natural language processing, there would be an explosion in the number of rules needed.

---

[2]ACFG rules are represented as LISP lists. The first element of the list corresponds to the left hand side of the rule. The rest of the elements correspond to constituents on the right hand side of the rule.

[3]Read as ''WH minus''; this follows the analysis now standard in generative grammar that declarative clauses bear the feature –WH (''minus WH'') and that interrogative clauses, whether they are content questions or yes-no questions, are +WH (''plus WH''). See Section 1.3.2.1 for more discussion of these two types of question.

[4]See Section 1.1.1.1 for discussion of subcategorization.

Because of this, the current version of the BBN ACFG does not store rules introducing lexical items, but rather generates them as needed by the parser on the basis of information stored in the lexicon and in conjunction with a morphology program that handles the regularly inflected forms[5]: such rules that are created on demand but not permanently stored are often referred to as "virtual rules". Thus, while the lexicon has no formal place in our system, it is used as a repository of lexical information (subcategorization, semantics, morphology, etc.) that is used to construct the virtual rules that the grammar actually uses.

1.1.1.1 Subcategorization

Virtual rules are also used to ensure that a member of a lexical category (currently V (Verb), N (Noun), and ADJ (ADJective)) appears with the correct *complements*. Complements are so called, in traditional grammar, because they "complete the meaning" of a lexical item in some way. For example, a transitive verb requires a noun phrase to follow it: "John fooled the boys" is grammatical but "*John fooled[6]" is not. Complements are lexically specified in that a given lexical item may or may not require (or permit) a particular category. Thus, intransitive verbs forbid a following noun phrase but may optionally permit other complements; e.g. "*The sun rose the boys" is ungrammatical, but "The sun rose over the mountains" is grammatical, although the phrase "over the mountains" is optional: "The sun rose" is grammatical as well.

The set of complements that a lexical item requires is often referred to as a *subcategorization frame*. While some formalisms, such as PATR-II [25], place most of the information about subcategorization in the lexicon and contain only a single rule for a category in which a lexical item takes complements—currently these are VP for V, N-BAR for N, and ADJ-BAR for ADJ—the BBN ACFG contains a rule for every subcategorization frame in which a lexical category can appear. Each such rule is "indexed", as it were, to a mnemonically named feature that must appear as the value of the subcategorization feature of any verb that can occur in that frame. (In this, it follows GPSG, which uses a similar indexing scheme.) The following two rules illustrate this aspect of the BBN ACFG formalism:

```
((VP (AGR :P :N) :NPTYPE :MOOD (AUXV (W (W (W (W (W (0AUX))))))
    (NOT-NEG)) (WH-) :TRX :TRX (-CONJ))
 (V (INTRANSITIVE :NPTYPE) :P :N :MOOD))

((VP (AGR :P :N) :NPTYPE :MOOD (AUXV (W (W (W (W (W (0AUX)))))))
    (NOT-NEG)) (WH-) :TRX :TRY (-CONJ))
 (V (TRANSITIVE :NPTYPE (TAKES-ACTIVE)) :P :N :MOOD)
 (NP :AGR (REALNP) (-POSS :POSSCLASS) (WH-) (OBJ (AGR :P :N)) :TRX
    :TRY :CONJC))
```

The first rule states that a VP may consist of a V (verb) followed by no other complements if the V bears the feature **(INTRANSITIVE)**. The second rule says that a VP may consist of a V followed by an NP just in case the verb is specified as being **(TRANSITIVE)** and capable of appearing in the active voice (**(TAKES-ACTIVE)**). As

[5] Irregularly inflected forms are listed in the lexical entry of their base form.

[6] * indicates ungrammaticality.

is usual, the variables :P, :N, :NPTYPE, and :MOOD are used to enforce agreement of the V and VP in these features.

1.1.2 Optional Elements

Currently there are no general mechanisms in the BBN ACFG that permit optional elements or that implement the Kleene star operator, which permits zero or more occurences of a specified element. To implement optionality and Kleene star, special categories are introduced that simulate these operations.

1.1.3 "Meta-rules" and Feature Value Default Mechanisms

Some annotated context-free formalisms, such as GPSG, include a mechanism that allows for rules that are, in some sense, predictable variants of other rules to be derived rather than being included in the object grammar. For example, GPSG provides a "meta-rule" facility, which, among other things, provides a means for deriving the passive version of transitive VP rules. The BBN ACFG does not have any such mechanism.[7]

A similar mechanism for compacting the size of grammars is some sort of feature defaulting mechanism, which would allow predictable features of one or more elements of a rule to be left unspecified. Currently, the BBN ACFG provides no such mechanism. It is very unlikely that the grammar formalism will itself provide such a facility, although it might be possible to provide an interface between the rules written and seen by users and developers and the form of the rules used by the parser.

1.1.4 Trace Flags

WH constituents, such as "who", "what", "how many ships", and "how long" are linked to an empty element (or trace) that appears in the position where the WH constituent is interpreted. For example, in the sentence, "who did John see" "who" is linked to an NP trace in the object position for the verb "see". In English, only one trace may appear in a single clause; compare:

who wonders what John gave Bill t_{what}
*who does Mary wonder what John gave t_{who} t_{what}

In the first case, "who" is interpreted as the subject of the matrix clause and "what" is interpreted as the object of the complement clause, so there is no more than one trace per clause and the restriction is satisfied. In the second case, "who" is linked to the indirect object position of "gave" (indicated by t_{who}) and "what" is linked to its direct object position (indicated by t_{what}), resulting in ungrammaticality. There are various linguistic and

[7]Some researchers who have tried to implement a Montague style compositional semantics using a GPSG syntax have reported that the meta-rule mechanism creates problems for the semantics. Thus, while not having meta-rules may increase the size of the object grammar, this may prevent problems in the area of semantics.

computational proposals to enforce this restriction. The one used in the BBN ACFG is that of difference lists, first suggested by Pereira [19].

1.2 The Parsing Algorithm

The algorithm used to parse the BBN ACFG is essentially that of Graham, Harrison, and Ruzzo [9], henceforth, GHR. This algorithm, in turn, is based on the familiar Cocke-Kasami-Younger (CKY) algorithm for context-free grammars. The original CKY algorithm could not be used to parse the BBN ACFG since that algorithm requires that a grammar be in Chomsky Normal Form (CNF), i.e. that each rule introducing non-terminal symbols—essentially the parts of speech, as opposed to the terminal symbols (lexical items and grammatical formatives)—be of the form

$$A \rightarrow B\ C$$

with exactly two non-terminal symbols on the right hand side. A grammar for a natural language will contain rules that deviate from CNF in the following ways:

rules with 0 symbols on the right hand side
: the rules that introduce traces, discussed above in Section 1.1.4, are of this type. Such rules are often called *empty rules*.

rules with only 1 symbol on the right hand side
: such as the rules introducing intransitive verb phrases, as in:

```
((VP ...)
 (V (INTRANSITIVE :NPTYPE) ...))
```

Such rules are often called *chain rules*.

rules with more than 2 symbols on the right hand side
: such as the rule introducing ditransitive verb phrases, as in:

```
((VP ...)
 (V (DITRANSITIVE (TAKES-ACTIVE)) ...)
 (NP ...)
 (NP ...))
```

Nevertheless, the CKY algorithm is quite simple and powerful: it starts with the terminal elements in a sentence and builds successively larger constituents that contain those already found and constructs all possible parses of the input. The GHR algorithm maintains this aspect of the control structure of the CKY algorithm without forcing the grammar to be in CNF. It does this by adding several mechanisms to CKY. All the chain rules of the grammar are collected into a special table that is consulted by the parser to determine if a chain rule is possible at any given point in the parse. All the empty rules of the grammar are collected into a similar table. Finally, for rules with more than two symbols on the right hand side, the mechanism of *dotted rules* is used. A dotted rule is like an ordinary rule, except that the right hand side is divided into two parts by a dot. This dot, in effect, makes the rule look as if it were in CNF. During the course of a parse, the parser will move the dot from the beginning of the right hand side of a rule to its end as the elements of the right hand side are found. Consider the following rule and its dotted rule equivalents:

$A \rightarrow B\ C\ D$

[Constituent A consists of the sub-constituents B C D]

$A \rightarrow .\ B\ C\ D$

[A rule that constructs an A; the parser has not yet found any of its sub-constituents]

$A \rightarrow B.\ C\ D$

[A rule that constructs an A; the parser has found a B and is now looking for a C and a D]

$A \rightarrow B\ C.\ D$

[A rule that constructs an A; the parser has found a B and a C is now looking for a D]

$A \rightarrow B\ C\ D.$

[A constituent of type A has been found]

The GHR algorithm will find all the dotted rules that derive an input sentence; this is another way of saying that it will find all the parses for a sentence. Looking at the parsing algorithm as a way of specifying all the grammatical word sequences of English, we may give the algorithm as in Figure 1-1.

The procedure used to build constituents out of previously found constituents does not involve simple matching but rather the process of *unification*, which matches the feature values in the different elements of a rule, as specified in the rule. As Section 1.1 showed, features may themselves be complex expressions, so that unification is a recursive process. Since the GHR algorithm, like the CKY algorithm, deals with context-free grammars, rather than context-free grammars annotated with features, the use of unification is an extension to the GHR algorithm; see [10] and [11] (included here as Appendix A) for full details. An important result reported in this work is that there is a class of ACFGs, called *depth-bounded ACFGs*, for which the parsing algorithm is guaranteed to find all parses and halt. Depth-bounded ACFGs are characterized by the property that the "depth" of a derivation, i.e. the number of non-terminal symbols that derive a terminal string, cannot grow unboundedly large unless the length of the string also increases. The fact that the parsing algorithm for this class of ACFGs halts is an important result, since unification grammars have the power of a Turing Machine and so, in the general case, cannot be guarenteed to halt. Moreover, the derivations ruled out by depth-bounded ACFGs are not needed for the analysis of natural languages, so this class of grammars is linguistically motivated, as well as computationally tractable.

```
for k = 1 to N
    for i = k-1 to 0 by -1
        dr[i,k] =
              (if i+1 = k
                  {(A → W. α) | W ∈ input[i,k])
               else
                  {(A → α B. β |
                    (A → α. B β) ∈ dr[i,j]
                     & (B → γ.) ∈ dr[j,k]}
               i < j < k
              )
              ∪
              {(A → B . β) | (B → α.) ∈ dr[i,k]}
              ∀ (A → B β) ∈ P
```

where

`N`	is the length of the input in words
`W`	is a variable ranging over terminal symbo.s (words)
`dr[i,k]`	is the set of dotted rules that span the input sentence from the *i*th through *k*th positions
`input[i,k]`	is the portion of the input sentence from the *i*th through *k*th positions
`P`	is the set of grammar rules (productions)

Figure 1-1: BBN ACFG Parsing Algorithm

1.3 Qualitative Measures of Coverage

This section describes the coverage of the current ACFG grammar in descriptive linguistc terms. Quantitative measures of coverage are presented in Section 1.4.

1.3.1 The Top Level Constructions

The BBN ACFG grammar currently handles the following types of utterances:

- Declarative sentences ("The Eisenhower is in the Indian Ocean")
- Interrogative sentences ("Is the Eisenhower in the Indian Ocean") in various types: the full spectrum is discussed in Section 1.3.2.1.
- Imperatives ("Display the Indian Ocean")
- NP utterances ("The ships in the Indian Ocean")
- Utterances made up of single interjections—single words or fixed phrases that constitute complete utterances ("Over and out", "Roger")
- Utterances made up of an interjection followed by a declarative clause ("Yes, the Eisenhower is in the Indian Ocean")
- Utterances made up of an interjection followed by an interrogative clause ("No, is the Eisenhower in the Indian Ocean")
- Utterances made up of an interjection followed by an imperative ("Yes, display the Indian Ocean")

1.3.2 Clausal Constructions

The current grammar handles clauses that comprise full utterances (so-called "matrix clauses") as well as subordinate clauses of different types.

1.3.2.1 Matrix Clauses

The following types of matrix clauses are currently handled:

- Declarative clauses ("The Eisenhower is in the Indian Ocean")
- Yes-no questions ("Is the Eisenhower in the Indian Ocean") and
  Content ("WH") questions ("Who is in the Indian Ocean")

  Content questions may involve various types of constituents:

  - Noun Phrases such as "who", "what", "how many ships", etc.
  - Adjective Phrases such as "how long", etc.
  - Locative and temporal expressions such as "where", "when", etc.
  - Adverbial expressions such as "how", "why", etc.

  Both yes-no and content questions involve a process of "subject-aux inversion" in which an "auxiliary element" and the subject are transposed; the ACFG grammar handles all such cases:

  - modals ("Must Eisenhower go to the Indian Ocean")
  - perfective "have" ("Has Eisenhower gone to the Indian Ocean")
  - progressive "be" ("Is Eisenhower going to the Indian Ocean")
  - passive "be" ("Is Eisenhower deployed to the Indian Ocean")
  - "main verb" "be" ("Is Eisenhower in the Indian Ocean")
  - auxiliary "do" ("Does Eisenhower have harpoon")

  The negated counterpart of each type is also handled:

- "Mustn't Eisenhower go to the Indian Ocean"
- "Hasn't Eisenhower gone to the Indian Ocean"
- "Isn't Eisenhower going to the Indian Ocean"
- "Isn't Eisenhower deployed to the Indian Ocean"
- "Isn't Eisenhower in the Indian Ocean"
- "Doesn't Eisenhower have harpoon"

Each of these question rules also allows ADVPs (ADVerb Phrases) of a specified type to appear after the subject NP:

- "Has Frederick ever gone to C3 on personnel readiness"
- "When was Eisenhower last in the Indian Ocean"
- "How soon will Wasp next chop to Atlantic Fleet from PACFLT"

1.3.2.2 Subordinate Clause Constructions

Subordinate clauses (that is, clauses that make up a subpart of a complete utterance) can be divided into two types:

Complement clauses
: These are clauses that are introduced as complements to lexical categories, such as verb, noun, or adjective, and which are permitted or forbidden by individual lexical items. For example, verbs like "believe" and "say" take complement clauses introduced by "that" while "go" and "come" do not.

Adjunct clauses
: These are clauses that are introduced in specified structural positions of phrases and clauses, independent of the exact lexical item that heads the phrase or clause. For example, noun phrases allow relative clauses introduced by WH words or "that" but do not permit clausal adjuncts introduced with "because"; compare "A/The man who was old came in" with "*A/The man because he was old came in".

Complement Clauses

The following types of complement clauses are currently handled:

- Finite clauses introduced by "that":

  1. Complement clauses that permit an optional "that" ("We believe that Eisenhower is in the Indian Ocean" and "We believe Eisenhower is in the Indian Ocean").

  2. Complement clauses that require "that" ("I order that Eisenhower sail to the Indian Ocean" but not "*I order Eisenhower sail to the Indian Ocean").

  These two types of complement clauses are often called indirect statements.

- Complement clauses that require "if" ("We wonder if Eisenhower is in the Indian Ocean").
- Complement clauses that require "whether" ("We wonder whether Eisenhower is in the Indian Ocean").
- Complement clauses that require a WH phrase ("We wonder who/which ship is in the Indian Ocean").

  These three types of complement clauses are often called indirect questions.

Adjunct Clauses

The following types of adjunct clauses are currently handled:

- Relative clauses introduced by a WH word ("Get the maximum speeds for carriers which are in Astoria") or "that" ("List carriers that are C5 on equipment").

  The current grammar also handles "extraposed" relative clauses with a WH word or "that" ("Five ships arrived that were C3").

The grammar also handles "stacked" relative clauses (i.e. multiple relative clause on the same head noun) ("Are there any ships which are located in China Sea that are C2") and conjoined relative clauses, even if one has "that" and the other has a WH word ("Give me a list of the carriers that are M3 on ASW and which are in New York"). In both these cases, where both "that" and a WH word appear, they may appear in either order.

- Clausal adjuncts that appear at the end of finite clauses ("Eisenhower is in the Indian Ocean because there is an emergency there") or infinitival complements ("We believe Eisenhower to be in the Indian Ocean because it is needed there").

1.3.2.3 Clausal Conjunction

In addition to conjunction and stacking of relative clauses, there is general clausal conjunction with "and" and "or".

1.3.2.4 The Immediate Constituents of Clauses

Clauses have as their immediate constituents (i.e. the elements that make up a clause) the subject noun phrase, a verb phrase (i.e. the verb and its complements) and various adjuncts. Currently, the following elements can appear as adjuncts:

- A participial clause ("Redraw the area updating the display")
- A participial clause introduced by a preposition ("Redraw the area without updating the display")
- Adverbial expressions of various types ("now", "yesterday", "on the twenty second of May")
- Locative and temporal expressions ("at seventy degrees north twenty three degrees east", "in the Indian Ocean")
- Prepositional phrases introduced by "with" that contain a noun phrase and a predicate phrase ("with areas off", "with system switches set to default values", "with the Shasta's in bright green")
- Extraposed relative clauses with "that" or a WH consitutent (see Subsection 1.3.2.2).
- Sentential adjuncts introduced by various complementizers ("after", "because", "unless", "until", etc).

1.3.3 Verb Phrase Constructions

As was explained in Section 1.1.1.1, lexical items are linked to the subcategorization frame(s) in which they can appear by means of a subcategorization feature whose values are associated with separate complement rules. For VPs, this is done by means of the argument **SUBCATFRAME** on Vs. There are currently 45 verbal subcategorization frames, which are derived from an extensive survey [15] of the literature concerning verb subcategorization in English (e.g. [1], [3], [20], [27], [28], [29], [30]). Since each of these features may be associated with more than one VP rule (e.g. the feature for transitive verbs is associated with both an active and a passive VP) there are more than 45 separate VP subcategorization rules; in fact, there are 69 such rules. Discusssion of these rules is beyond the range of this section but see [16] for a detailed discussion of each.

1.3.4 Auxiliary Constructions

The BBN ACFG provides the full range of standard VP auxiliary phenomena in English. Since there are no meta-rules in the formalism (see Section 1.1.3) there is a separate rule for each type of auxiliary element:

- modals (''Eisenhower must go to the Indian Ocean'')
- perfective ''have'' (''Eisenhower has gone to the Indian Ocean'')
- progressive ''be'' (''Eisenhower is going to the Indian Ocean'')
- passive ''be'' (''Eisenhower is deployed to the Indian Ocean'')
- ''main verb'' ''be'' (''Eisenhower is in the Indian Ocean'')
- auxiliary ''do'' (''Eisenhower does have harpoon'')

There is a fixed order to auxiliary elements in English:

- modal elements—which include *can*, *could*, *may*, *might*, *must*, *shall*, *should*, *will*, *would*, as well as *'ll* as a contraction for *will*—precede
- perfective *have*, which precedes
- progressive *be*, which precedes
- passive *be*, which precedes
- main verbs

In addition, ''main verb'' *be* follows progressive *be* and is in complementary distribution with passive *be* and lexical main verbs.

The following sentence shows this order:

You must have been being bad or they wouldn't have gotten angry at you.

While this order is fixed, not all auxiliary elements need to appear all the time; as long as they appear in the correct order, any subset of them can appear:

Eisenhower must leave.
Eisenhower must have left.
Eisenhower must have been leaving.
Eisenhower has left.
Eisenhower is leaving.
Eisenhower must be leaving.
...

In addition, ''dummy'' *do* can only appear if no other auxiliary element appears. It appears in questions, in negated sentences, and in emphatic statements:

Did Eisenhower reach the Indian Ocean?
Eisenhower didn't reach the Indian Ocean.
Eisenhower did reach the Indian Ocean!

English also contains restrictions on the placement of the sentence level negative elements *not* and its contracted form *n't*. These negative formatives appear following the first auxiliary element in a clause. If there is no other auxiliary element in the sentence, *do* appears, since negation cannot appear with a simple main verb in English:

*John went not/wentn't.
*John not/Johnn't went.
vs.
John didn't go.

The ACFG grammar handles both the order and optionality behavior of the English auxiliary system. It also handles the placement of negation.

1.3.5 Noun Phrases

Noun phrases consist of the following elements (where parentheses indicate optionality):

(Determiner) (Comparative/Superlative-Adjectives) (Positive-Adjectives) N-Bar (Adjuncts)/(Relative-clauses)

Determiners include those elements typically called determiners: the definite and indefinite articles (*the*, *a*, *an*), demonstratives (*this*, *that*), quantifiers (*all*, *some*, *each*, ...), etc. Determiner structure is discussed in more detail in Section 1.3.5.1.

Comparative and Superlative Adjectives
are introduced into noun phrases separately from positive adjectives for both syntactic and semantic reasons. The semantic argument is simple: it is difficult or impossible to write a semantic rule that will work for an arbitrary number of adjectives that will work for both positive and superlative adjectives. There are several syntactic facts that also point to non-positive adjectives being introduced separately from positive adjectives:

- Superlative and comparative adjectives must precede non-positive adjectives and cannot be intermixed with one another:

  the fastest green ships
  *the green fastest ships

- Superlative and comparative adjectives preferentially select particular noun phrase adjuncts. Comparative adjectives can co-occur with phrases introduced by *than*:

  a faster ship than the Frederick

  Superlative adjectives can co-occur with phrases introduced by *of*:

  the fastest ship of the unit

  Positive adjectives impose no such restrictions.

- Superlative adjectives require definite determiners, while positive adjectives impose no such restrictions.

Positive adjectives are the base (non-compared) forms of adjectives (''green'', ''fast'').

An unlimited number of adjectives (positive, comparative, or superlative) are introduced via the optionality mechanism mentioned in Section 1.1.2.

N-BAR introduces the head noun of the noun phrase together with its complements, if any. N-BAR rules are discussed in some more detail in Section 1.3.5.2.

Adjuncts Currently, there is no recursive structure to Noun Phrase adjuncts, i.e. only one adjunct can be introduced at a time, but this will probably change. Currently, the following constituents can appear as adjuncts:

- Adjective Phrases: These are Adjective Phrases either headed by adjectives that lexically specify a postnominal position (such as *left* in ''the fuel left'' or adjective

phrases containing a complement, since adjective phrases containing complements cannot appear prenominally in English; compare ''sufficient fuel'', ''fuel sufficient to get there'', ''*sufficient to get there fuel''. For more discussion of adjective placement, see Section 1.3.6.

- Participial VPs: ''the ship approaching port''
- Passive VPs: ''the ship deployed to the Indian Ocean''
- Prepositional Phrases introduced by *with*, *for*, or *of*: ''all ships with harpoon'', ''their arrival for the meeting'', ''equipment of the highest quality''.
- Locative and temporal expressions: ''the meeting on the seventeenth of July''.

Relative clauses — Relative clauses with relative *that* (''all ships that are C1'') and WH words (''all ships which are C1''), including ''whose'' (''commanders whose ships are C1'') are handled.

Conjoined relative clauses may also appear here. *That* relatives may be conjoined with WH relatives. ''Stacked'' relative clauses, such as ''ships that are in the Indian Ocean that are C1'', may also appear.

1.3.5.1 Noun Phrase Determiners

The determiner structure of English noun phrases is one of the most complicated areas of English syntax. There are a great number of dependencies between the determiner and the classes of nouns that appear as the head of NP. The determiner itself may have a complex structure with similar dependencies among its elements.

The ACFG grammar ensures that ''count nouns'' (i.e. nouns that can appear with articles, that can be pluralized, etc.) appear only with count determiners, such as ''every'' (''every man'' vs. ''*much man''), etc.; and that non-count nouns (both ''mass'' and ''abstract'' nouns) (i.e. nouns that typically do not appear with articles, do not easily pluralize, etc.), appear only with non-count determiners such as ''much'' (''much sand'' vs. ''*every sand), etc. It also allows other determiner-noun combinations, such as proper nouns that permit ''the'' to appear before them, including ship names (e.g. ''The Eisenhower''), place names (e.g. ''the Bering Straits''), etc.

The ACFG grammar allows the full range of determiners, including quantificational determiners, such as ''every'' and ''how many'', numerical determiners, including ordinals, cardinals, and fractions, such as ''three'', ''twenty one'', ''three quarters'', as well as the ''pre-determiner'' elements that appear with them, such as ''more than'' (''more than six ships'', ''more than three quarters of the ships'') or ''all'' (''all six ships''), the traditional articles ''the'', ''a'', and ''an'', the demonstratives ''this'', ''that'', ''these'', and ''those'', the possessive determiners, which include both ''possessive pronouns'' (''my'', ''your'', ''their'', etc.) and full scale possessive NPs, as well as other determiner elements, such as ''both'', ''either'', ''neither'', etc. Other, specialized determiner constructions are also handled, such as ''the/my last three vacations''.

Some determiners can appear without any following noun, such as ''those'' (as in ''All the ships in Diego Garcia are C4 but those in Honolulu are C1'') and such constructions are also handled.

The ACFG grammar also handles ''question'' determiners, such as ''which'' and ''how many'' and partitive determiners, such as ''some'' and ''each'' (e.g. ''some of the ships'', ''each of the ships'').

1.3.5.2 Complements to Nouns

N-BAR is the constituent that contains the head noun of a noun phrase and its complements. There are currently only 12 subcategorization frames for nouns in the grammar; this reflects the nature of linguistic knowledge about complements to noun. While complements to nouns are less numerous than those to verbs, correspondingly less is known about them. In addition to ordinary complements to nouns, N-BAR also contains a limited form of noun-noun compounding, as in "readiness rating", "combat readiness", and "combat readiness rating".

1.3.5.3 Noun Phrase Conjunction

Currently, there are grammar rules for conjunction of two or more noun phrases with "and" or "or" as well as rules for conjunctions of the form "both NP and NP" and "either NP or NP".

1.3.5.4 Specialized Noun Phrase Constructions

In addition to the general noun phrase constructions already discussed, there are also specialized noun phrase constructions. Currently, these include:

time expressions in any of the following forms:

fifteen hundred
fifteen hundred hours
fifteen hundred hours zulu
fifteen hundred zulu

Time expressions may also be introduced by prepositions, such as "at".

date expressions in any of the following forms:

twenty December
December twenty
December twentieth
twentieth December
twentieth of December
the twentieth December
the twentieth of December

as well as date expressions consisting only of the day of the week, e.g. "Wednesday". Date expressions may also be introduced by prepositions, such as "on".

year expressions in either of the forms:

nineteen eighty seven
eighty seven

latitude and longitude expressions
in any of the following forms:

one hundred degrees north fifty degrees east
one hundred degrees north and fifty degrees east
one hundred degrees fifty minutes north and fifty degrees forty five minutes east
one hundred degrees fifty minutes north fifty degrees forty five minutes east
a latitude of one hundred degrees north and a longitude of fifty degrees east
latitude one hundred degrees north longitude fifty degrees east
latutude one hundred degrees north and longitude fifty degrees east

title expressions these include both true title expressions, as well as analogous expressions:

Admiral George Metaxas
Admiral Metaxas
USS Enterprise
U.S.S. Enterprise

1.3.6 Adjective Phrase Constructions

Adjectives may appear alone and may also be modified by a special class of adverb: such as "very" or "how". Other, more specialized Adjective Phrase constructions are also handled:

<ordinal number> <superlative adjective>
e.g. "second best"

<numerical NP> <comparative adjective>
e.g. "three miles closer"

<comparative adjective> than NP
e.g. "faster than the Wasp"

<comparative adjective> than S
e.g. "faster than the Wasp is"

<comparative adjective> PP *than* NP *is* PP
e.g. "farther from Guam than the Wasp is from Diego"

<comparative adjective> than NP PP
e.g. "closer than the Wasp to port"

less <positive adjective> than NP
e.g. "less truthful than John"

less <positive adjective> than S
e.g. "less truthful than John is"

as <positive adjective> as NP
e.g. "as fast as the Wasp"

as <positive adjective> as S
e.g. "as fast as the Wasp is"

1.3.6.1 Complements to Adjectives

Like verbs and nouns, adjectives also take complements. The complexity of the complement system to adjectives is between that of nouns and verbs. Currently, there are only 11 subcategorization frames, but others will be added. As was the case with the complement system to nouns, there are fewer complement types to adjectives, but less work has been done in this area.

1.3.6.2 Adjective Phrase Position

Adjective phrases can appear in three different positions (excluding constructions where they appear as complements): pre-nominally, as in "the C3 ship"; post-nominally, as in "the fuel remaining"; and in predicate position, as in "the ship is C3". Most adjectives can appear in pre-nominal and predicate position (as "C3" illustrates) but there are some adjectives that can only appear predicatively ("the man is afraid" vs. "*the afraid man") or post-nominally ("presents galore" vs. "*galore presents" or "*the presents are galore"). In addition, even adjectives that can normally appear pre-nominally appear post-nominally when they occur with a complement: compare "a faithful servant", "a servant faithful to his master", "*a faithful to his master servant". The ACFG grammar and lexicon handle adjective phrase position correctly.

1.3.6.3 Adjective Phrase Conjunction

Currently, there are grammar rules for conjunction of two or more adjective phrases with "and" or "or".

1.3.7 Adverbial Constructions

The following types of adverbial constructions appear in the current grammar:

temporal expressions
: such as "now" and "when" as well as temporal NPs.

date expressions
: including date NPs.

manner expressions
: currently only "how".

"reason" or purpose expressions
: currently only "why".

Adverbial expressions can appear in three positions in a clause: initially, as in "Currently, the Eisenhower is in Diego"; medially, as in "Eisenhower never went to C5"; and finally, as in "Eisenhower is in the Indian Ocean now". Not all adverbs can appear in all three of these positions and the ACFG grammar and lexicon contain features to ensure that adverbs appear only in the appropriate positions.

Currently, adverbial expressions only appear in clause medial and final position; the clause initial position will be added later.

1.3.8 Other Constructions

The ACFG grammar also handles prepositional phrase constructions and locative and temporal expressions.

1.4 Quantitative Measures of Coverage

1.4.1 Grammar Size

The current ACFG grammar contains 672 rules; of these, 369 introduce grammatical formatives (such as the articles "a", "the", prepositions, etc.) The remaining 303 rules handle the grammatical constructions of the language.

1.4.2 Syntactic Coverage

Syntactic coverage is measured using sentences from the Resource Management domain. These sentences are divided into two subsets: a training corpus of 791 sentences—this corpus is used as a basis for grammar development; and a test corpus of 200 sentences—these sentences are run through the parser "blind" (i.e. without any person actually seeing them) to see how well coverage on the training corpus generalizes to previously unseen but related material. The grammar currently covers 85 percent of the training corpus and 64 percent of the test corpus. Figure 1-2 shows the figures for training and test corpus coverage for July, 1987 and October, 1987, as well as the current coverage.

1.4.3 Perplexity

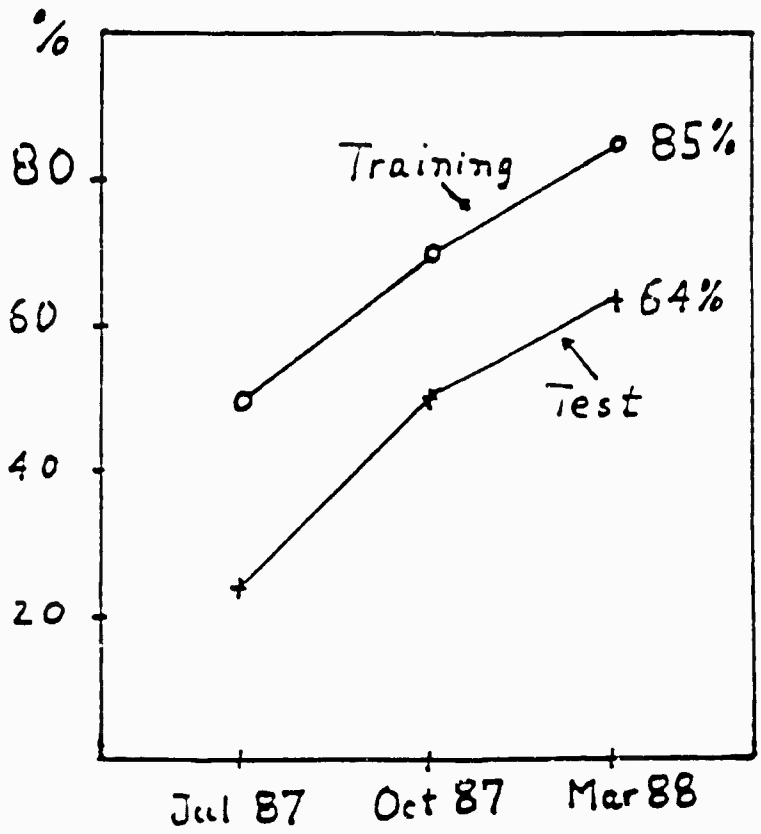
For spoken language systems, *perplexity* is an important measurement. Perplexity is defined as follows:

$$Q = 2^H$$

where Q is perplexity and where H is

$$H = \frac{-1}{n} \log P(w_1, \ldots, w_n)$$

where $w_1, \ldots, w_n$ represents a set of possible input sentences stacked as an n-long text (typically from a corpus). Intuitively, given some prefix (initial substring, possibly null) of a sentence, perplexity represents the number of possible word choices that are allowed to follow within a given grammar. Here are some sentences from the Resource Management corpus with their associated perplexity measurements in the ACFG grammar:



	July, 1987	October, 1987	March, 1988
Training	50%	71%	85%
Test	22%	50%	64%

Figure 1-2: BBN ACFG Resource Management Training and Test Corpus Coverage

```
[1] Show me  the  Indian Ocean.
    630   583 612 583    1
[2] Display thirty degrees south seventy degrees east.
    630     588  613     31    28       10       4
[3] Where are the  frigates and carriers?
    630    10 583 583     655 593
[4] What is    the  readiness of   Eisenhower?
    630  655  641 583       655  610
[5] When will Eisenhower be   C1?
    630   10  583          607 647
[6] What is    Frederick's     readiness?
    630  655  641        649  583
[7] Which ships are  faster than Frederick?
    630    654  691 641   641  583
[8] Display Frederick's     track.
    630     588        616  583
```

The overall perplexity of these sentences is 314. Some comments about the individual perplexity figures is in order. First, it should be noted that the possessive morpheme *'s* is treated as a separate element of the input sentence by the parser. This explains why there are more perplexity measures in examples [6] and [8] than there are surface phonological words. Second, while perplexity is normally high for this grammar, perplexity becomes reduced in several cases. One of these is the case of collocations—lexical items that consist of more than one orthographic word, such as "Indian Ocean" in example [1]. For the 1000-word Resource Management task, the word "Indian" only appears as the first element of "Indian Ocean"; hence, perplexity drops to 1 after "Indian". Another case is that of subgrammars, such as that for latitude/longitude expressions, as seen in example [2]. At the end of the latitude/longitude expression, word choice is down to 4, since only the cardinal compass directions are possible at that point. Finally, there are certain positions where a general grammar for English allows only a limited number of possibilities. For example, after the adverbial question words "where" in example [3] and "when" in example [5], perplexity drops to 10. This is because such question adverbials unambiguously trigger subject-aux inversion (discussed above in Section 1.3.2.1). Hence, the only elements that can follow these question words are the auxiliary elements: the modals and the inflected forms of *be*, *do*, and *have*.

1.4.4 Ambiguity

As the coverage of the grammar increases, two types of ambiguity also increase. First, at any given point in a sentence, there will probably be more possible choices that must be tried. Second, the number of parses that a sentence has may increase. Information on the number of parses per sentence has been collected for the 128 test corpus sentences that the current grammar handles. The number of parses per sentence shows the following distribution:

Parses per sentence:	Number of sentences so parsed:
1	35
2	36
3	15
4	19
5	3
6	6
8	3
10	1
12	6
16	1
18	1
29	1
48	1

Here is the sorted version of the same list:

Parses per sentence:	Number of sentences so parsed:
2	36
1	35
4	19
3	15
6	6
12	6
5	3
8	3
10	1
16	1
18	1
29	1
48	1

This distribution shows a mean of 4 (rounded from 3.875), a median of 2.5, and a mode of 2.

There are various reasons why a sentence may be treated as ambiguous by the parser. First, it may be the case that a reasonable grammar for English will produce multiple syntactic interpretations for a given sentence, though semantic or pragmatic information may decide among them. This type of ambiguity might be considered irreducible. Another type of ambiguity may simply be the result of errors in the grammar, either because a syntactic phenomenon has not been handled correctly or because of a typographic error. Such mistakes are corrected when detected. The third source of ambiguity is more problematic. These are cases where the formalism in which the grammar is written introduces certain types of ambiguity that cannot be eliminated by rewriting existing rules or adding new rules. For examples, noun compounds such as "supplies readiness" or "data screen" are $n \times m$ ways ambiguous, where n is the number of possible grammatical number readings of the first member of the nominal compound (for example, "data" is treated as both singular and plural in the current grammar) and m is the number of subcategorization frames associated with the first member. In some cases, it may be possible to eliminate the ambiguity by changes to the formalism. For example, in the present case, the addition of a disjunction mechanism, which would collapse all the separate readings into one, would probably do the job.

1.4.5 Overgeneration

Another area in which the performance of the current grammar can be measured is that of *overgeneration*. Overgeneration results from a looseness of the grammar that results in it producing utterances that are not English.[8] Overgeneration may be divided into two types, analogous to the linguistic notions of weak generative capacity and strong generative capacity [5]. Weak generative capacity refers to the set of sentences that a grammar generates (or accepts). Strong generative capacity refers to the structures that a grammar assigns to the sentences that it generates (or accepts). For example, consider two (rather trivial) grammars that both generate the single sentence "The Wasp is deployed." These two grammars are identical in weak generative capacity. However, if the first grammar assigns this sentence the structure:

[The Wasp is] deployed

while the second assigns it the structure

The Wasp [is deployed]

they differ in strong generative capacity.

We can now define analogous subtypes of overgeneration: a grammar weakly overgenerates if it accepts non-English utterances as English;[9] while a grammar strongly overgenerates if it accepts English sentences but assigns them the incorrect structures.

Consider the sentence: "Show all ships." The grammar currently assigns it two structures:

Show [all ships]

where there is a single noun phrase object, and also:

Show [all] [ships]

where *all* is taken to be the indirect object (like *me* in "Show me the ships") and *ships* is treated as the direct object. Note that allowing *all* to be a complete noun phrase does not generate non-English structures since *all* can be a complete noun phrase (as in "All were ready"); however, it does admit parses which are not intended, as in the present example.

In addition to such cases, there is currently one area known in which the grammar strongly overgenerates by assigning an impossible structure to an English sentence; this is the case of ellipsis of possessive noun phrases, as in:

Frederick's speed is greater than Eisenhower's

The current grammar assigns such sentences a parse in which the possessive marker *'s* is interpreted as the reduced form of the verb *is*. However, as is well known (see, e.g. [2]), it is impossible for verbs in English to reduce in such comparative constructions. Compare:

---

[8] In the context of a parser, we might refer to this as over-acceptance: the parser accepts input strings that are not English sentences.

[9] This type of overgeneration is directly related to, and inversely correlated with, perplexity.

He is smarter than you are.
*He is smarter than you're.

Note that the current grammar would accept the second example, so that the treatment of verb reduction weakly overgenerates in addition to strongly overgenerating.

(In passing, it should be added that the current grammar also assigns the example the parse:

Frederick's speed is greater than Eisenhower's (speed)

(where () enclose the elided element) which is the intended parse.)

Aside from the treatment of verb contraction, which introduces both strong and weak overgeneration, there are no known hard and fast cases of weak overgeneration in the current grammar. Consider the utterance:

More was Frederick's position.

which is accepted by the current grammar. While this is not a likely sentence of English, the quite acceptable:

That was Frederick's position.

uses the same set of rules. Thus, while the acceptance of the previous utterance might appear to be a case of syntactic overgeneration, it is clear that the utterance is not syntactically ill-formed: rather, it is semantically anomalous. Currently, all the known cases of accepted utterances that might initially seem to be instances of overgeneration, except for the spurious verb contraction examples, fall into the class of utterances that are not syntactically ill-formed.

1.5 Future Plans

This section describes the work that is currently planned to increase the coverage and performance of the syntactic portion of the system.

1.5.1 Extending Coverage

As was stated in Section 1.4.2, the current grammar covers 85% of the Resource Management training corpus and 64% of the test corpus. The training sentences that do not parse have been divided into classes based on the syntactic issues that they raise and ordered on the basis of the ease of adding the phenomenon and the expected increase in coverage: i.e. easier changes that increase coverage greatly are ordered before more difficult changes that do not result in large increases in coverage and also before more minor changes that result in small increases in coverage. The following is an illustrative sample of the full set of classes.

- **Add modifiers to Adverb Phrases:**
  Redefine time window for the chart of Eastern Taiwan to start *four days sooner*
  How fast could Yorktown get to seventy eight north forty east

- **Increase the range of prepositions introducing time and date phrases:**
  Find all data for the Meteor updated *since twenty three hundred hours*
  Get me deployments of submarines *during eighty four*
  Find subs that were in the Mozambique Channel *as of tenth of January*
  Couldn't Camden arrive in port *by tomorrow*

- **Add "last" and "next" as pre-modifiers to date phrases:**
  Get me *last month's* casreps for Hepburn
  List *last week's* casreps from Davidson
  Get posit data for cruisers employed before *last eighteen January*
  Is the Peoria due in port before twenty four hundred zulu *next Wednesday*

- **Increase the range of comparative and superlative expressions:**
  Are there three LAMPS cruisers with maximum speeds *not greater than fifteen knots*
  Are there six vessels that are in Kodiak with readiness *more than C4*
  Do any carriers that are in Midpac have *more fuel than it*
  Is Flint at *three quarters of fuel capacity or less*
  Has Citrus been at sea *the longest of submarines which are in Bering Strait*
  Is the Vandergrift's displacement *less than average for all M1 ASUW subs*
  Is Jarrett's gross displacement *smaller than average for the south Persian Sea ships*
  Is the Fanning's fleet ID *the same as Downes's*

- **Add distance expressions:**
  Is the Confidence *more than a kilometer from Conifer*
  How soon can Tripoli get to *within nine kilometers of the Sherman*
  Is Willamette *within seven miles of seventy four west forty north*

- **Add use of color adjectives as nouns:**
  Show Sherman's track in *dim yellow* with the Shasta's in *bright green*
  Set color of the Independence's track to *bright red*
  Use *bright green* for tracks of nuclear surface ships

- **Add multiple adjuncts to NPs and allow both relative clauses and adjuncts to appear:**
  Draw the seven subs *which are in Gulf of California with the lowest fuel capacity*
  Display the four ships *in the Formosa Strait with the largest fuel capacities*
  What's the number of submarines *that are in east Atlantic Ocean without Tacan*

- **Add adverbs not currently in the system:**
  Are more than five cruisers *currently* in home port
  Make chart *again* in low resolution

- **Add extraposed adjectival complements in NPs:**
  Does Ajax have *sufficient* fuel *to get to Port Victoria*
  Does the Bainbridge have *enough* fuel *to arrive at his destination*

- **Add general "the" deletion:**
  Never mind *next chart display*
  Show the chart of Mozambique Channel with the Davidson displayed *in center*
  Don't draw the chart on *redraw*

- **Miscellaneous changes that only add one or two sentences each:**
  Add agent *by* phrase to passive sentences:
  What training problem was reported *by Camden* last month
  Add use of "a" for "one" in numbers:
  Is there a frigate in the Gulf of Thailand longer than *a thousand* meters
  Add use of *-ing* participles as NPs:
  How much does *including Friday's data* change that figure
  What would it be *counting only C2 carriers*
  Add limited cross-categorial conjunction:
  Find me all the submarines *in south Bering Sea and that are M3 on ASW*

1.5.2 Reducing Spurious Ambiguity

Since the program for collecting coverage information for the training corpus also provides information about the number of parses per sentence, it is possible to note sentences with a large number of parses, examine the parse trees assigned to them, and modify the grammar to reduce the number of parses, where the grammar is overgenerating. At times, this has resulted in dramatic savings: "Why was Citrus's MOB mission area changed thirty one September" had its total number of parses reduced from 54 to 2 with changes in two grammar rules. While this process will undoubtedly continue in the future, it is labor intensive, since all the parses for an ambiguous sentence must be examined by hand, so we have sought more general procedures that can reduce ambiguity but which are also linguistically motivated. The most promising scheme is to assign probabilities to grammar rules. Once probabilities are assigned, we can explore the following strategies, among others:

1. Find all parses for a given input, but ignore all parses that use grammar rules with a probability below a fixed threshold.
2. Find all the parses of a given input that involve only grammar rules with a probability above a fixed threshold; parses involving rules with probabilities below this threshold will not be constructed.
3. Use probabilities to arrange the rules of the grammar into tiers; all parses for a given input involving rules from all tiers will be found, but parses involving rules from lower probability tiers will be ignored.
4. Use probabilities to arrange the rules of the grammar into tiers; in parsing a given input, if at least one parse is found using rules from the tier with the highest probability, parses involving rules from tiers with lower probabilities will not be constructed. However, if no parse is found at the highest tier, rules from successively lower tiers will be used, until at least one parse is found.

Finally, in addition to the general strategies for reducing ambiguity sketched here, we have arrived at a mechanism which eliminates the ungrammatical acceptance of *'s* as the reduced form of the verb *be* in comparative construction, discussed above in Section 1.4.5.

1.5.3 Changes to the Grammar Formalism

As one more technique for eliminating spurious ambiguity, we will explore the possibility of using a simple disjunction mechanism to collapse the readings assigned to noun-noun compounds, discussed above in Section 1.4.4.

2. The Semantic Component

2.1 Introduction

This chapter describes the semantic component of the BBN Spoken Language System. This semantic component operates upon the output of the syntactic component (the parser) in such a way as to give the correct response to a user's request. The body of the chapter is as follows.

Section 2.2 gives theoretical background to the work described here. In particular, it presents our view of the nature of semantic knowledge, which comes from the perspective of logical model theory.

Section 2.3 presents the architecture of the semantic component, which is based on the notion of "multi-level semantics" [4] [22] in which a user utterance is assigned meaning through successive translation from one level of representation to another.

Section 2.4 presents the logical language used to represent meaning in the BBN Spoken Language System.

Section 2.5 illustrates the semantic processing of the system by tracing the translations of an example question in some detail.

Section 2.6 discusses the underlying system and various utilities that support the abstractions of the logical language.

Section 2.7 discusses our implementation accomplishments over the past year and gives direction for future work.

2.2 The Nature of Semantic Knowledge

Semantics is traditionally seen as the part of linguistics which attempts to account for the relation between expressions of language and their meanings. The next Question—what is a meaning—is a philosophical one and need not concern us unduly. We will consider that to know the "meaning" of a declarative sentence like "The door is closed" means to know what sort of state the world would have to be in for that sentence to be true. These requirements on the state of the world are called the "truth-conditions" of the sentence, and this notion of meaning is called the "truth-conditional" theory [26].

The truth-conditional theory does not leave out other sentence types in any necessary way. A question like "Is the door closed?" is seen as being about the truth or falsity of the corresponding statement ("The door is

closed''). A command like ''Close the door!'' is seen as being a command to make the statement ''The door is closed'' true. Finally, a question like ''What is on the table?'' is seen as asking for values of ''x'' for which the statement ''x is on the table'' is true.

A corollary of the truth-conditional view is that if we know the truth-conditions of a sentence, and if we have complete knowledge of the state of the world, then we can say correctly whether that sentence is true or false.

Obviously it does not make sense for this knowledge to be represented as a table which, given any state of the world, pairs sentences with the value 'TRUE' or 'FALSE'. As Chomsky [5] has shown, the set of expressions in any natural language (and for that matter, many artificial ones as well) is infinite. Modern grammatical theory sees this infinite set as being generated by a recursive system consisting of a finite number of basic expressions and a finite number of recursive formation rules which build expressions out of other expressions.

For instance, the following is a toy example of such a recursive system, generating a trivial but still infinite language:

Basic Expressions: 'FREDERICK', 'INDIAN-OCEAN', 'IN'

Formation Rule: if α,β, and γ are expressions then

$\alpha(\beta,\gamma)$

is an expression as well

This system produces an infinite set of expressions, among whose members are the following expressions:

IN(FREDERICK,INDIAN-OCEAN)

IN(INDIAN-OCEAN,FREDERICK)

FREDERICK(IN,INDIAN-OCEAN)

IN(IN(FREDERICK,INDIAN-OCEAN),INDIAN-OCEAN)

....

An *interpretation* of a language is a systematic assignment of values to the expressions of the language. We think of an interpretation as a function—call it 'F'—which takes an expression of the language as its argument and returns the value assigned. The notions of truth and falsehood simply correspond to the values 'TRUE' and 'FALSE'. Given the way that expressions are built up out of other expressions as above it seems reasonable to wonder whether the assignment of values to entire expressions does not depend in some way on the assignment of values to their parts.

The *Principle of Compositionality* is a theory of this dependence. It states that the value of an expression is a function of the values of its parts and of the mode of combination of its parts. A compositional interpretation is constituted as follows:

1) a value is assigned to each Basic Expression of the language
2) an interpretation rule is assigned to each formation rule of the language

The interpretation rules compute the values of expressions from the values of their ''parts''—the expressions the formation rule combined.

The following is an example of a compositional interpretation for the language we defined above. Let the assignments of values to terminals be:

FREDERICK => s_1
INDIAN-OCEAN => o_1
IN => {<<s_1,o_1>,TRUE>,
<<s_1,s_1>,FALSE>,
<<o_1,o_1>,FALSE>,
<<o_1,s_1>,FALSE>}

It is important to note that the 's_1','o_1','TRUE','FALSE' above are not to be thought of symbols of any language but rather as ''things themselves''—be they ship, ocean, truth or falsehood. If I could, I would place the actual objects on the page but typography has its limits. The angle brackets '<' and '>' serve to delimit ordered pairs.

Now the interpretation rule for expressions of the form $\alpha(\beta,\gamma)$ is as follows, where α',β', and γ' are the translations of α, β and γ respectively:

if α' is a set containing an element e_0 such that e_0 is an ordered pair whose first element is the ordered pair $<\beta',\gamma'>$ then the semantic value of $\alpha(\beta,\gamma)$ is the second element of e_0

It might help to think of this rule as doing ''table look-up'' on the semantic value α'.

We have now specified an interpretation of the language. This interpretation assigns the semantic value 'TRUE' to the expression

IN(FREDERICK,INDIAN-OCEAN)

and the semantic value 'FALSE' to the expressions

IN(INDIAN-OCEAN,FREDERICK)

IN(FREDERICK,FREDERICK)

IN(INDIAN-OCEAN,INDIAN-OCEAN)

The interpretation assigns no semantic value at all to any other expression of the language, including:

INDIAN-OCEAN(IN,FREDERICK)

FREDERICK(INDIAN-OCEAN,IN)

IN(IN(FREDERICK,INDIAN-OCEAN),INDIAN-OCEAN)

....

The value assigned to an expression is usually called its ''denotation''; expressions for which no value is assigned are termed ''denotationless''.

We can perhaps now see how to deal with the problem posed above—namely, how to determine the truth or falsity of a statement in English given a complete state of the world. To each word of the language, such as the proper noun ''Frederick'' or the preposition ''in'', we assign an appropriate semantic value, where the domain of this assignment is the whole set of things in the world: ships, oceans, numbers, people etc. An appropriate assignment of values to words is one consonant with our knowledge about the state the world is in at the time the

assignment is made. For example, the assignment to "in" would relate certain pairs of physical objects and locations to the value TRUE and other such pairs to the value FALSE according to whether or not the physical object was in that location or out of it.

For each rule of English grammar, we assign an appropriate rule of semantic interpretation. An appropriate pairing of semantic and syntactic rules means one consonant with our intuitions about the meaning of expressions in our language.

Of course there are difficulties in the way of this proposal First, very many English utterances are syntactically ambiguous: there is more than one way to built them up using the rules of a grammar. Secondly, many English words, such as "bank" have more than one meaning, and thus cannot be assigned a single semantic value.

Because expressions of English are so ambiguous we find it advantageous to first translate them to an unambiguous language—a language of logic. For example, the statement "Frederick is in the Indian Ocean" could be translated:

IN(FREDERICK,INDIAN-OCEAN)

We have already seen how to give an interpretation for this expression. A question like "Is Frederick in the Indian Ocean" could be translated:

QUERY[IN(FREDERICK,INDIAN-OCEAN)]

Any ambiguity of an English expression corresponds to multiple translations into this logic.

So far we have left out of consideration the fact that the state of our world—and therefore the truth-value of the statements we make—varies with time. The ship Frederick might be located in the Indian Ocean at time t and then sail out of it at a later time t'. It would be possible to accommodate such time-dependency quite straightforwardly by adding an extra argument to 'IN', making it a three-place instead of a two-place predicate. This would ignore the fact, however, that time in natural-language expressions is frequently supplied by *context*—for example, the time at which a statement is made or a question asked, a time which is not mentioned in the expression itself. Such contextual dependency on time argues strongly for making the interpretation of expressions contextually dependent on time as well.

To do this the interpretation function F is given an extra argument for time. This argument for time is referred to as the *index* of interpretation. Other contextual factors might also be incorporated as other indices—among them, the notion of a "possible world". We speak of the "intension" of an expression as a function from such indices to the value—the "extension"—that the expression has at those indices [18].

2.3 System Design Overview

The semantic component of the BBN Spoken Language System is divided into several processing stages. Each produces as its output an expression of a logical language; all but the first also take an expression of a logical language as their input. The idea of semantics by translation, mentioned in the previous section, is expanded here to encompass not one but several translations in a series.

The first stage accepts as input the output of the parser; this takes the form of a parse tree, whether of a complete sentence or some grammatical constituent thereof. It uses as a knowledge base a set of semantic translation rules paired one-for-one with the rules of the syntactic grammar, together with a set of semantic entries, one for each word of the lexicon, which pair words with expressions of a logical language called EFL, for English-Oriented Formal Language. Using the semantic rules and lexical entries it recursively builds up an EFL translation of the parsed utterance.

The lexicon pairs each word with just one expression of EFL, no matter how many senses that word may have. For this reason, the EFL level of representation is still ambiguous. A sentence like "Bill reached the bank" would have only the single EFL translation:

```
PAST[Reach'(Bill',The(Bank'))]
```

even though it has (at least) two different senses—one in which Bill reaches the bank of some river and another in which he arrives at a bank building.

The EFL expression above is not the direct result of translation of the parse tree to EFL, but a simplified form of it. After the translation to EFL, a simplification step is performed. The simplification uses a fixed set of logical simplification rules, and is performed after every translation between levels.

After this simplification, the EFL expression is translated to to one or more expressions of a second logical language, WML, for World Model Language. A set of translation rules are used in this step. These translation rules map each constant expression of EFL to one or more expressions of WML. All possible combinations are constructed; those which are "anomalous" (in the sense to be defined in the next section) are filtered out. If no translations survive, the system classifies the utterance as grammatical but not meaningful in its subject domain and the user is so informed.

The constant symbols of WML correspond to the primitive concepts of the subject domain. The choice of WML constants is governed by two requirements: 1) for any index, specifying the extension of each constant gives the complete "state of affairs" in the subject domain at that index; and 2) no constant has its extension completely determined by the extensions of any other constant or set of constants. The set of WML constants thus serves as a complete but minimal "domain model".

In the third stage of processing an expression of WML is translated to an expression of DBL, for Data Base Language. Just as above this translation process is driven by a knowledge base of translation rules relating WML

constants to DBL expressions, the difference being that each WML constant is related to only a single DBL expression. The constant symbols of DBL correspond to the data files of the information system that natural language queries are addressed to. These data files can be seen as actually specifying the denotations of the DBL constants, thus grounding the model theoretic semantics of the utterance in the underlying comptuer system itself. If a particular WML constant happens not to have a DBL translation, the query containing it is meaningful in the subject domain but not answerable by the system, and the user is so informed.

The reason for having separate WML and DBL levels is not just the possibility of incomplete information just now alluded to, but also the fact that the requirement of efficiency for the storage of data tends to produce compact tabular structures that do not correspond one for one to the natural concepts of the subject domain. Translation is therefore needed.

Finally we have the evaluation stage, when the "value" of the DBL expression is computed against the current state of the data files. The answer is represented in yet a fourth language, CVL for Canonical Value Language. This is the stage of processing at which failures of existential presupposition are detected and reported to the user. An example would be the query "Which carriers in the IO are C3" when it happens that there currently *are* no carriers in the IO. If there are no such presupposition failures the CVL answer representation can then serve as input to whatever process generates the display of the result according to particular pragmatic goals.

The languages EFL, WML and DBL are simply different instantiations of the same logical language, differing from each other only in the set of constant symbols each of them includes. The same set of formation rules generates the expressions of each. The exception is CVL, whose formation rules are simply a small subset of the formation rules of the others.

This is a useful fact, since it allows the same set of logical simplification rules to be used on expressions of all four languages. After each processing stage but the last (which is in some sense the ultimate simplification) the simplifier is invoked to produce a smaller, simpler expression. The simplification transformations express necessary truths of logic, such as '$P \wedge TRUE = P$' and are applied by an iterated, recursive descent algorithm.

In the next section I present the higher-order intensional logical language which serves as the representational framework for the foregoing.

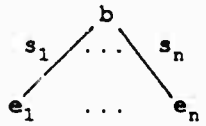
2.4 The Logic

Expressions of the logic are divided into three groups: constants, variables, and complex expressions. Complex expressions are built up by the formation rules from constants, variables, and other complex expressions.

Expressions of the logic are labeled trees in which both nodes and branches are labeled. Constants and variables are the terminal nodes. The node labels are called "branching categories", the branch labels, "selectors". Each branching category 'b' is associated with a set of selectors 'F-SELECTORS(b)'. If 'b' is a branching category, e_1,.... expressions and $\{S_1,.....S_n\}$ = F-SELECTORS(b) then the following is an expression:

$$\langle b,\ \{\langle S_1, e_1\rangle, \ldots, \langle S_n, e_n\rangle\}\rangle$$

or drawing it in tree form:



An example would be the branching category APPLY whose selectors are {FUNCTION,ARGUMENTS}. Let 'FREDERICK' and 'READINESS-OF' be constant symbols. Then the following is an expression:

```
<APPLY, {<FUNCTION, READINESS-OF>, <ARGUMENT, FREDERICK>}>
```

Certain branching categories have a special selector, 'VAR', as one of their selectors. Let 'FORALL' be a branching category whose selectors are {VAR,RESTRICTION,FORMULA} and let 'X' be a variable, 'P' and 'SHIPS' constants. Then the following represents an expression which is said to bind the variable 'X'

```
<FORALL, {<VAR, X>, <RESTRICTION, SHIPS>,
          <FORMULA, <APPLY, {<FUNCTION, P>,
                             <ARGUMENT, X>}>>}>
```

Obviously this structural notation is cumbersome to write. Therefore, an external form of the language is provided. The above would be written:

```
(FORALL X SHIPS (P X))
```

Each expression of the logic has a *type*, which constrains both the types of other expressions with which the expression can meaningfully combine and the values the expression can assume. There is an infinite set of types, enumerated by a set of recursive rules. The base of the recursion is a finite set of atomic types, varying from instantiation to instantiation but including at least the set INTEGERS, REALS, WORLDS, TIMES, STRINGS, SPEECH-ACTS, TRUTH-VALUES and NULL-SET. For reasons that we shall see shortly, these are called the *formal* types.

The following are some of the formation rules for types. We assume that

every atomic type is a type

if α, α_1,.....α_n and β are types then the following are also types:

SETS (α)
FUN (α, β)
TUPLES (α_1, . . . , α_n)
UNION (α_1, . . . , α_n)

are assigned "domains", which are sets of semantic values representing the possible denotations which expressions having that type may take on. Formal types are special in that their domains are fixed for all allowed interpretations of the language. The domains of atomic types are mutually disjoint. The relation SUB-TYPE? and the binary operation TYPE-INTERSECTION are defined for types. Note that for distinct atomic types α and β, TYPE-INTERSECTION(α,β) = NULL-SET.

As an example, '1' and 2' are constants of type INTEGERS, while '2.0' is a constant of type REALS and 'TRUE' and 'FALSE' are constants of type TRUTH-VALUES. The symbol '+' is a constant of type

FUN (TUPLES (UNION (REALS, INTEGERS) , UNION (REALS, INTEGERS))
UNION (REALS, INTEGERS))

The types of complex expressions are computed by a rule associated with their branching category, a rule which takes as input the types of the sub-expressions at their branches. Let the symbol 'FREDERICK' be a constant of type 'SHIPS' and let 'READINESS-OF' be a constant of type FUN(SHIPS,R-VALUES), where 'R-VALUES' is also an atomic type. Then the construction:

<APPLY, {<FUNCTION, READINESS-OF>, <ARGUMENT, FREDERICK>}>

is a meaningful expression of type 'R-VALUES'. Now, suppose 'RONALD-REAGAN' is a constant of type PERSONS'. The construction

<APPLY, {<FUNCTION, READINESS-OF>, <ARGUMENT, RONALD-REAGAN>}>

is not a meaningful expression, and has the type NULL-SET. It can never have a denotation under any indices, and accordingly has the empty set as the set of values it can take on.

We have now put into place the machinery needed to distinguish between meaningful and meaningless expressions. Meaningless expressions have NULL-SET as their type, and can never have a denotation at any index. Meaningful expressions may fail to have a denotation at certain indices —consider "The King of France" but nonetheless have a denotation at *some* index.

The infinite hierarchy of types allows more complex functions than we have seen so far—functions on sets for instance. This power turns out to be necessary for many English utterances. Consider a sentence like "The boys gather". Unlike the similar sentence "The boys walk", it is not true that each boy, individually, gathers—this simply wouldn't make sense. Rather, it is a predication that one makes of the set of boys as a whole. To represent it, one needs a predicate which is applied to sets of people, or a function constant of type:

FUN (SETS (PEOPLE) , TRUTH-VALUES)

If 'GATHER' is a function constant of this type, then the utterance "The boys gather" can be represented as:

GATHER (BOYS)

This is a small illustration of the power and flexibility of the infinite type system of the language. Examples still

more complex than the verb "gather" do not give us trouble. If we like, we can have constants whose types are sets of sets of sets, or functions from functions to functions.

So far we have been content to simply introduce a new constant whenever we have need. This procedure quickly becomes inconvienent when we want to construct the representation of a notion which is built up from others. Consider a function constant 'LOVES' whose type is:

```
(FUN (TUPLES PEOPLE PEOPLE) TRUTH-VALUES)
```

and an individual constant 'MARY' whose type is 'PEOPLE' and which translates the name "Mary" in our lexicon. Now suppose we want to consider a special property, the property of loving Mary. We wouldn't want to have to come up with a new constant symbol, say 'LOVES-MARY', for every possible object of affection in the domain of discourse. What we really would like to do is have some way of constructing an expression representing this property out of materials—constant symbols—already to hand.

The branching category 'LAMBDA' gives us a way of doing this. Its selectors are {VAR,RESTRICTION,BODY}. The property of loving Mary is then expressed by:

```
(LAMBDA X PEOPLE (LOVES X MARY))
```

The type of this expression is a function type from the type of its VAR, or bound variable, (PEOPLE) to the type of its BODY (TRUTH-VALUES). This is simply a predicate.

Now suppose we wanted to say that a particular person, say Bill, loves Mary. Let "Bill" be translated by an individual constant 'BILL' of type PEOPLE. We can then apply the above LAMBDA expression as we would any other predicate, to obtain the expression:

```
((LAMBDA X PEOPLE (LOVES X MARY)) BILL)
```

This expression can be simplified to an equivalent expression by a process called Lambda-reduction. If the argument (here, 'BILL') is of a type "appropriate" to that of the bound variable of the lambda-expression, it can be substituted for that bound variable in the body of the lambda-expression. (The rule is actually more complicated, renaming free variables in the argument that would otherwise become bound in the substitution, and not substituting for variant occurences of the bound variable that are bound again in the body.) The above can be reduced to:

```
(LOVES BILL MARY)
```

The LAMBDA branching category plays a very important role in our system. The translation rules that map between the EFL, WML, and DBL levels use lambda-expressions to construct representations in the target language that are equivalent to notions in the source language. In this way a mapping can be constructued between expressions of the two languages even though they have different sets of constant symbols.

The next section of this chapter shows how the logic is used by the system to construct a meaning represenatation of a sentence.

2.5 Example of Processing

In this section we go through an actual example of processing, showing how the representation of the meaning of an utterance is progressively transformed through the various levels of the system.

We'll take for our example the question "Is Frederick in the Indian Ocean?". The parse tree for this question is shown in Figure 2-1, which is a hard-copy of an actual screen-display. (Note that it goes from left to right as opposed to from top to bottom.) The left-hand button of the mouse shows the syntactic rule associated with a given node, the middle the semantic rule, and the right-hand button the EFL interpretation of the node. A right-hand button click on the left-most node "START" gives the EFL interpretation of the entire sentence.

Let's get some idea of the recursion. Clicking right on the node labeled "N" just to the left of the nodes labeled "INDIAN" and "OCEAN" gives the following EFL expression:

```
INDIAN-OCEAN
```

The phrase "Indian Ocean" is a collocation, which the system considers as being a single word. Clicking middle on the node labeled "LOC-TEMP-PHRASE" gives us the semantic rule:

```
(meta-lambda ($prep $np)
 (lambda x anytype ((Q $np) (lambda y anytype ($prep x y)))))
```

The 'meta-lambda' is an operation which takes a list of 'meta-variables (those prefixed with the "$" sign) and an expression containing meta-variables. The meta-variables are the inputs to this rule; when regular logic-expressions are plugged in for them in the body, the EFL semantic interpretation of the node is produced.

Clicking right on this node gives the EFL interpretation which is the result of this rule plus the EFL interpretations of the children of the node:

```
(lambda x anytype ((Q (setof indian-ocean))
                   (lambda y anytype (in x y))))
```

Note that the preposition "in" is represented by the descriptive EFL constant "IN", underlined above.

What the system does with the entire sentence can be seen by clicking right on the node labeled "START". This produces the expression:

```
(QUERY
 ((INTENSION
   ((LAMBDA*T*
     ((LAMBDA P ANYTYPE (P DONTCARE))
      (LAMBDA TRVAR ANYTYPE ((LAMBDA X ANYTYPE (EXTENSION X))
                             (INTENSION ((Q (SETOF FREDERICK))
                                         (LAMBDA X ANYTYPE ((Q (SETOF INDIAN-OCEAN))
                                                            (LAMBDA Y ANYTYPE (IN X Y))
                                                            )))))))
     *T*))
   NOW ACTUAL-WORLD))
```

This expression can be simplified in various ways. Simplification is accomplished by the function SIMPLIFY, which is called between levels of translation. The simplified version of an EFL translation can be obtained from this display by clicking right while holding down the "control" key. The result of simplification is:

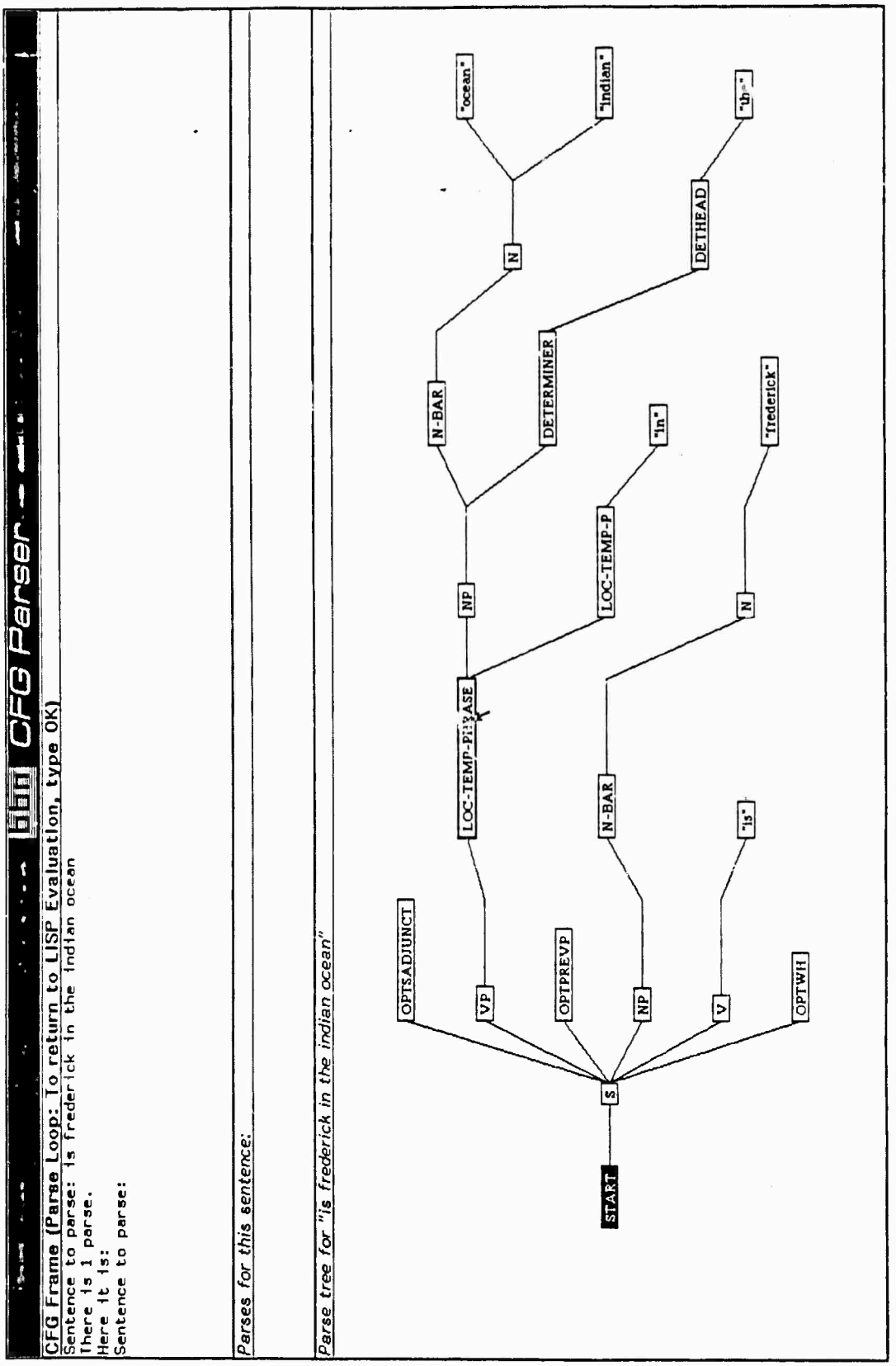


Figure 2-1: Screen Display of Parse Tree

(QUERY ((INTENSION (IN FREDERICK INDIAN-OCEAN)) NOW ACTUAL-WORLD))

The next step is to translate the EFL expression to WML. Recall that this translation step is an ambiguous one which can result in more than one WML expression. In this particular case there is just one ambiguous element in the sentence: the preposition "in". The word "in" can mean many things in different contexts: consider "Frederick is in the Indian Ocean" vs. "Frederick is in the Pacific Fleet". Accordingly, the descriptive constant IN is an ambiguous symbol, and has the following set of WML translations.

1. (lambda (X Y) ((groups ships) (groups fleets)) (forall x1 (set x2 ships (INC x2 x)) (exists y1 (set y2 fleets (INC y2 y)) (equal (fleet-of x1) y1))))
2. (lambda (x y) ((groups things) (groups places)) (forall x1 (set x2 things (INC x2 x)) (exists y1 (set y2 places (INC y2 y)) (sub-location (location-of x1) y1))))

The first of these corresponds to the "in-fleet" reading, the second to the "in-place" reading. They are typed in terms of "groups" of ships, fleets, places etc. to handle cases of plural arguments to "in" such as we find in the NPs "ships in the fleet", "ships in the fleets etc. (These matters are discussed in the paper [23]

The set of WML translations for the entire sentence is assembled from all possible combinations of the translations of ambiguous EFL constants. A filtering step is applied to the translation which excludes WML translations with incompatible combinations. This filtering can be applied to intermediate results of the recursive translation. Translation (1) above can filtered away because 'frederick' is a ship and ships and fleets have no common members.

The simplified WML translation of the entire sentence is a follows:

(QUERY ((INTENSION (SUB-LOCATION (LOCATION-OF FREDERICK) INDIAN-OCEAN)) NOW ACTUAL-WORLD))

This is the unambiguous meaning representation of the sentence "Is Frederick in the Indian Ocean?".

2.6 The Semantic Framework System

2.6.1 Introduction

The semantic framework system implements the abstractions of our logic in software. It is an interconnected system of modules and definitions which allow other programs—specifically the various translation modules sketched in Section 2.3—to be written in terms of these mathematical abstractions without concern for implementation details. In consequence the translation modules themselves are quite simple, and comprise only a few pages of code. Translation modules and indeed the entire processing architecture can be quite easily modified as the need arises.

The semantic framework system includes:

- Data structure support for logical language expressions

- A means for easily extending the logic to include new operations
- Declaration functions for specifying descriptive constants
- A type system to compute expression types
- A sub-system for performing translations and transformations on expressions
- Syntax and consistency checkers to prevent errors in KB data entry

We now proceed to discuss each of these in detail.

2.6.2 Logical Expressions as Data Abstractions

We have implemented the logical language with LISP functions in which branching categories and selectors are represented as LISP atoms and constructions as implementation-dependent data-structures.

We have the function CONSTRUCTION:

CONSTRUCTION (*branch-category branches*)

where the argument *branches* is an assoc list pairing selectors and expressions or, in the case that the selector is a multi-branching, LISP lists of expressions. It creates and returns a tree data structure. The following constructs a universal quantification expression which claims for all CRUISERS that the predicate P holds:

```
(CONSTRUCTION 'UNIVERSAL-QUANTIFICATION
              '((FORMULA APPLY P (VARIABLE X SHIPS))
                (VAR VARIABLE X SHIPS)
                (RESTRICTION . CRUISERS)))
```

We have the function BRANCH-CATEGORY:

BRANCH-CATEGORY (*expression*)

which takes a tree and returns its branching category. If it were applied to the example just presented it would return the atom UNIVERSAL-QUANTIFICATION.

There is the function SELECTION:

SELECTION (*expression selector*)

which takes a tree data structure and appropriate selector, and returns the sub-tree structure pointed to. Suppose the construction returned above to be bound to the atom FOO. Then the following:

```
(SELECTION FOO 'VAR)
```

would evaluate to '(VAR X SHIPS)'.

Also available are functions (actually macros) which evaluate only some of their arguments. These are very handy when selectors and branch category of the expression is already known when code is written. There is the function MAKE, whose form is:

```
(MAKE <bc>
      <sel1> <exp1>
        ....
      <seln> <expn>)
```

where only the <expi> are evaluated. There is also the function SELECT, whose form is:

```
(SELECT <selector> <exp>)
```

where only <exp> is evaluated.

2.6.3 Functions for Defining Constants

The following functions are used to define constants and types:

dt (*name language &optional def*)
dft (*name*)

dc (*name language type*)
dfc (*name type*)

The above stand for, respectively, "declare type", "declare formal type", "declare constant", "declare formal constant". The *name* and *language* arguments can only be filled by Lisp symbols.

The *language* argument must be supplied for non-formal types and constants, and must be a declared language name. The list of language names is bound to the atom **language-names**; to add a language name, one adds to this list.

2.6.4 Functions for Extending the Language

The semantic framework system has a unique and powerful feature: it directly supports the extension of the logical language used for semantic representation.

Branching category declarations are separated into two classes: those declaring regular branching categories and those declaring branching categories of the type sub-language. The following are their argument patterns:

DEF-BC (*branch-category selector-list &optional typerule eval-rule*)

DEF-TYPE-BC (*branch-category selector-list &optional eval-rule*)

None of these arguments are evaluated. The *branch-category* argument must be a LISP atom, and the *selector-list* argument a list of atoms. This argument specifies the value of the function F-SELECTORS for the given branching category.

The *typerule* argument specifies a rule for computing the type of the expression of the branching category from the types of its sub-trees. At the current time this "rule" is just a Lisp-code lambda expression. In the near future a more sophisticated notation will be used.

The *eval-rule* argument is also optional, and analogously specifies a rule for computing the *value* of the expression from the values of its sub-trees. Eval-rules are Lisp code around which has been wrapped either a form '(EVAL-T *)' or a form '(EVAL-O *)'. "EVAL-T" stands for "evaluate transparently". A rule of this kind is supplied with the values of the sub-trees of the expression. "EVAL-O" stands for "evaluate opaquely". A rule of this kind is supplied with the sub-trees themselves, and must itself take on the responsibility for how they are evaluated. Such rules are inherently non-compositional, and are (currently) used only for branching categories which bind a variable.

2.6.5 Translations and Transformations

A number of functions are provided that facilitate the transformation of one logic expression into another.

We distinguish two different kinds of transformation: local and global. In a local transformation, only constant symbols are transformed. Branch-categories are left unchanged and thus the structure of the input expression carries over to the output expression. An example of this kind of transformation would be the WML to DBL translation.

The function TRANSLATE:

TRANSLATE(exp ct)

takes an expression and a function. The function argument takes a constant and returns an expression (whether another constant or a complex lambda-expression) which is that constant's translation. The algorithm is quite simple and is given below:

```
TRANSLATE(exp ct) =def
    BC <- BC(exp)
    Selectors <- F-selectors(BC)
    (if: BC = CONSTANT
     then: CT(exp)
     elseif: BC = VARIABLE
     then: exp
     else: (construction BC
                         (pairlis selectors
                                  (for s in selectors
                                       collect TRANSLATE(exp.s CT)))))
```

Global transformations are unconstrained. They take the form '<input pattern> => <output pattern>'. Patterns are implemented by the notion of meta-expressions, distinguished from regular expressions by the appearance of meta-variables, which are atoms prefixed by "$". An example global transformation would be:

```
(ELEMENT-OF $x (SETOF $a)) => (EQUAL $x $a)
```

which states an equivalence between two expressions, where the right-hand side is simpler than the left.

The function **MATCH**(*pattern,exp,env*) tells whether or not the expression 'exp' matches 'pattern'. 'Env' is an assoc list pairing meta-variables and expressions: it is ordinarily NIL when match is called at top-level.

The output of MATCH is either an association list pairing meta-variables and expressions or the atom FAIL. Thus:

```
(MATCH (ELEMENT-OF FREDERICK (SETOF VINSON))
       (ELEMENT-OF $x (SETOF $a))
       nil)
```

evaluates to the assoc-list '(($a . VINSON)($x . FREDERICK))

This output is then given to the function which handles the right-hand side of a global transformation. This function is **META-EVALUATE**(*exp,env*). Its effect is take an expression containing meta-variables, and an environment which assigns those meta-variables, and return the resulting instantiation. As an example:

```
(META-EVALUATE '(EQUAL $a $x)
               '(($a . VINSON) ($x . FREDERICK)))
```

returns the expression '(EQUAL $a $x)'.

META-EVALUATE is used in the initial semantic interpretation to EFL, where the semantic rules are like the right-hand-sides of global transformations.

A function DEFTRANSFORMATIONSET is provided for defininining arbitary sets of global transformations. The function APPLY-TRANSFORMATION-SET applies such a set of transformations, working in a recursive descent fashion. The function APPLY-TRANSFORMATIONS-REPEATEDLY applies a given set of transformations over and over again, until no more can be applied. It is the basis for the function SIMPLIFY.

2.6.6 Functions for Comparing Types

There are two major functions for comparing types. These are SUB-TYPE?, which computes the subsumption relation between types, and TYPE-INTERSECTION, which takes two type expressions and returns a third which is the "largest" sub-type of both.

For any two types t1, t2, and for all indices of evaluation,the following statement is true of SUB-TYPE?

```
SUB-TYPE?(t1,t2) -> DEN(t1) ⊆ DEN(t2)
```

This simply says that if t1 is a sub-type of t2, it is always the case that the denotation (i.e. the domain) of t1 is a subset of the denotation of t2. Note that the converse is *not* true: that is, if the denotations of one type expression is a subset of another, even at all indices of evaluation, it is not necessarily the case that the first expression is a subset of the second.

The algorithm which computes SUB-TYPE? is now presented. In order to deal with union types we will first need the function COMPONENTS, which takes a type expression and returns the set of basic type expressions that make it up:

```
COMPONENTS(type) =def

    if   type = NULL-SET
    then {}
    else if   BC(type) = UNION
         then compute-union(for: t
              in:  type.sets
              take: COMPONENTS(t))
         else {type}
```

where 'compute-union' takes a set of sets and returns the set that is the union of these sets. Note that the distinguished type NULL-SET has an empty set of components as is appropriate for it.

The following are example results of COMPONENTS when applied to various type expressions:

```
COMPONENTS(DESTROYERS) -> DESTROYERS
COMPONENTS(UNION(A,B,C)) -> {A,B,C}
COMPONENTS(UNION(A,UNION(B,B),A) -> {A,B}
COMPONENTS(FUN(A,B)) -> {A,B}
```

We can now present the algorithm for SUB-TYPE?:

```
SUB-TYPE?(T1,T2) =def

  if T1 = NULL-SET
  then TRUE
  else if T2 = NULL-SET
       then FALSE
       else for: x
            in: COMPONENTS(T1)
            holds: for: y
                   in: COMPONENTS(T2)
                   exists: COMPONENT-SUB-TYPE?(x,y)
```

where COMPONENT-SUB-TYPE? is defined by:

```
COMPONENT-SUB-TYPE?(T1,T2) =def

    BC1 <- BC(T1)
    BC2 <- BC(T2)
    if BC1 = CONSTANT and BC2 = CONSTANT
    then T1 = T2
    else if BC1 = BC2 and BC1 ε {L,B,SETS,F}
         then SUB-TYPE?(T1.ELEMENT-TYPE,T2.ELEMENT-TYPE)
         else if BC1 = BC2 and BC = TUPLES
              then for: x
                   in: T1.ORDERED-ELEMENT-TYPES
                   as: y
                   in: T2.ORDERED-ELEMENT-TYPES
                   holds: SUB-TYPE?(X,Y)
```

2.6.7 Syntax Checkers for Logic Expressions

Obviously it is not convenient for humans to use the constructor functions to write down logic expressions. For this reason, an external form of the logic is provided for their use. This external form is a LISP s-expression, which the function PARSE-EXP turns into an internal form expression, checking for errors or omissions as it does so. If errors are found appropriate messages are printed on the user's terminal and the function returns NIL to its callers.

Errors include improper syntax for branch-categories, use of undeclared symbols, and use of forbidden symbols (such as branch-category names) as terms. Both regular and meta-variable expressions are parsed.

Since the conversion from external to internal form is always required, PARSE-EXP is invoked at every knowledge-base entry point. Any transformation, translation or rule that does not pass PARSE-EXP is simply refused, and the user so notified. In this way the system is protected from a great many errors that would otherwise only appear at run-time.

Another function is provided to check whether the type of expressions is meaningful. Called CHECK-EXP-TYPE, it prints out any anomalous constituent.

Finally, it is often necessary to turn internal form back into external form for user readibility. The function PPL does the conversion and the function SHOW-EXP does conversion and pretty-printing.

2.7 Accomplishments over the Last Year

2.7.1 Implementation Status

We have implemented the semantic processing architecture this report describes and used it in demonstrations in July and October of 1987.

We have used the tools described in the last section in constructing the knowledge bases of the multi-level semantics system. In particular, we have implemented:

- A set of 468 structural semantic rules for the syntactic rules of the grammar (70% coverage)
- A set of 629 Semantic entries for the words in the lexicon (87% coverage)
- A "domain model" of 97 type and 589 descriptive constant declarations
- A collection of 114 logical simplification transformation rules
- A set of 51 EFL to WML translation rules

Semantic coverage currently stands at 30% of sentences parsed.

2.7.2 Theoretical Issues and Publications

In the process of building these knowledge bases we have had to contend with a number of semantic problems for which no widely accepted solution exists. One of these is the problem of "relational" nouns, such as "speed", "brother", etc. which, in contrast to regular "categonal" nouns such as "man", "ship" and "elephant", seem to require an argument for their reference to be made clear. These "arguments" do not correspond, however, to syntactic relations on the noun.

We have developed and implemented an approach to relational noun semantics in which the arguments are supplied by semantic and not syntactic means. This involves treating relational noun denotations not as sets of individuals, but as sets of ordered pairs corresponding to the extension of a relation.

A paper describing this approach has been accepted for publication and presentation at the forthcoming 1988 meeting of the Association for Computational Linguistics [7].

Another semantic problem has to do with the combination of parts of speech taking arguments—verbs, adjectives, prepositions, as well as relational nouns—with plural arguments. In this case the issue is how the argument-taking item is to be "distributed" over the individual members of the denotation of the plural noun phrase. As has been earlier shown by Scha [21], the way in which this is done depends on the particular lexical item in question. We have developed and implemented an approach to this problem which uses the distinction between the levels of EFL and WML to translate an initial "collective" application on the EFL level to its final "distributed" WML counterpart.

A paper describing this work has been accepted for publication and presentation at the forthcoming 1988 meeting of the Association for Computational Linguistics [23].

2.7.3 Future Work

Having put into place the general framework this chapter describes we plan to spend the coming year increasing the coverage on the Resource Management Corpus. This will involve focussing on a number of semantic issues raised in the corpus, including:

- The interaction of tense and time adverbials
- The treatment of generic and mass nouns
- Intrasentential Anaphora
- "Time-series" perspectives on concepts, e.g. "Vinson's last five locations"

3. Speech and Natural Language Integration

In the preceding chapters, we have described the natural language components of the BBN Spoken Language system: the parsing algorithm that uses the BBN ACFG (Chapter 1) and the semantic interpreter that derives a meaningful interpretation of text input (Chapter 2). In this chapter, we will describe our approach to integrating syntax and semantics with acoustic scoring for speech understanding.

The goal of speech understanding is to determine what was spoken and the corresponding meaning of the input utterance. To achieve optimal performance, i.e., the maximum correct understanding rate, we need to find the most likely word sequence consistent with syntax and semantics. This poses the problem of a large search space which must be explored judiciously so that an utterance can be processed in a reasonable amount of time with reasonable computational resources.

There are several possible approaches to solving the speech understanding problem.

One possible approach, which we have demonstrated previously, is the serial connection. In this approach, speech recognition and natural language processing are performed serially and independently, with the speech recognition component computing the best scoring answer using acoustic models and its own language model, and then passing the answer to the natural language component for processing and interpretation. The critical problem with this approach is the possibility of a mismatch between the speech language model and the natural language grammar: the sequence of words recognized by the speech component could possibly fall outside the coverage of the natural language grammar, causing the system to break down altogether. Also, if the speech recognition component makes an error, there is little chance for recovery. Therefore, to have any chance of success, one needs to fully integrate speech and natural language, where integration means using same the language model to jointly perform speech recognition and natural language understanding in a single search space.

One approach to integration is to compile the natural language syntax and semantics into a single network such as a Finite State Automaton (FSA) or a Recursive Transition Network (RTN) appropriate for performing a top-down time-synchronous search to find the best hypothesis [6]. However, this assumes that such a network can in fact be built from the declarative unification grammar formalism of our natural language syntactic component and our semantics component. A close examination of our grammar reveals that the number of equivalent context-free rules needed to make an FSA network from our unification grammar (semantics not included) would run into the hundreds of thousands, and the number of arcs in this FSA network would be many times that size. No computer on the market or on paper today would have a virtual address capacity anywhere near this size, not to mention the paging penalties that would be incurred even if such a computer were available.

The approach that we have taken, then, is to perform parsing (in the natural language processing sense) on the speech input. This approach consists of a two step process. First, the speech component computes a very dense word lattice: all words that are plausible acoustically anywhere in the input utterance would be computed, with a separate score for every starting and ending time. Given this word lattice, the natural language component can search for the most likely meaningful sentence as a path through the lattice.

As such, the problem can now be posed as a parsing problem solvable by parsing algorithms similar to the text parsing algorithm described in Chapter 1. Whereas the text parser takes one sentence as input, the "speech parser" takes the lattice of alternate word hypotheses, and finds in the lattice *all* grammatical (syntax only) sentences and assigns each sentence an acoustic likelihood score. Henceforth, we shall call this speech parser the Word Lattice Parser. To handle the notion of scoring in parsing, we have extended the text parser to deal with acoustic likelihood scores. Every grammatical constituent (starting from the terminals) now has an acoustic score attached to it, indicating the likelihood of this constituent occurring across a particular time interval of the input utterance. Parsing now means matching the rules in the grammar as well as performing dynamic programming (DP) of input speech using acoustic models of the word. The final answer is a complete grammatical constituent (the **(START)** symbol in the grammar) spanning the entire utterance that has the highest acoustic score.

This chapter is organized as follows: in Section 3.1, we discuss the speech component of our speech understanding system that is used to compute the acoustic scores for the words in the vocabulary; in Section 3.2, we give details of our integration of speech and syntax; in Section 3.3, we describe how we currently incorporate semantics to find the best interpretation of the input; and finally in Section 3.4, we discuss some of the system implementation issues in building the integrated BBN Spoken Language System (SLS) for speech understanding.

3.1 Speech

For the purpose of integration, the speech component needs to compute the acoustic likelihood scores for all words in the vocabulary between any time intervals i and j. We define the acoustic score of a word W to be the logarithm of the conditional probability:

```
S(i,j|W) = LOG(Prob(i,j|W))
```

where $Prob(i,j|W)$ is the score or likelihood of the hypothesis that terminal or word W produces the observed input acoustic data between times i and j. The acoustic data is typically a sequence of analyzed and vector-quantized (VQ) input spectra sampled every 10 millisecond [6]. We model the input speech at the phonetic level using robust context-dependent Hidden Markov Models (HMM) of the phoneme [24]. The acoustic model for each word in the vocabulary is then derived from the concatenation of these context-dependent phonetic HMMs.

Using these acoustic models of the word, one can compute the acoustic scores for each word on the input utterance using a nonlinear time alignment procedure. A computationally efficient method is to use the trellis algorithm [17]. We use backward (in time) trellis computation to compute all scores $\{Prob(i,j|W): 0<i<j\leq T\}$ in a single pass for a fixed time j, as shown in Figure 3-1.

In this version of the algorithm the index i only needs to range from j minus 1 down to j minus the maximum duration for the word, which in our system is based on the number of phonemes in the word. We further improved on the computational efficiency by switching the order of the W loop and the i loop. This allows us to do time-synchronous pruning similar to that described in [24] among all words that ended at time j. This pruning

```
;; For all ending times j
for j = 1, T   do

    ;; For all words in the vocabulary
    for W in {W}   do

        ;; For all beginning times i compute Prob(i,j|W)
        for i = j-1, Max(0, j-MaxDur[W]) by -1 do

            Perform within-word DTW and compute Prob(i,j|W)
```

where DTW is performed using the trellis algorithm.

Figure 3-1: Dynamic Time Warping (DTW) algorithm 1

algorithm compares words that end at the same time j and eliminates those that score poorly acoustically relative to the best scoring word for all time $i,i<j$. It has given us a factor of two or more reduction in the acoustic computation. The improved algorithm is shown in Figure 3-2.

```
;; For all ending times j
for j = 1, T   do

    ;; For all beginning times i compute Prob(i,j|W)
    for i = j-1, 1 by -1 do

        ;; For all words in the vocabulary
        for W in {W}   do
        if (i < j-MaxDur[W])
           quit

           else

             Perform within-word DTW and compute Prob(i,j|W)
             Also keep track of maximum score across words and
             perform time-synchronous pruning.
```

Figure 3-2: Dynamic Time Warping (DTW) algorithm 2

The computational complexity of this backward DTW algorithm is proportional to $2\times J^2\times T$, where J is the maximum duration of a word and T is the length of input utterance in frames. The result of performing this acoustic computation is a word lattice which is then used for the integration of speech and the natural language components.

3.2 Integration of Speech and Syntax

Previously, we argued for the need to integrate speech and natural language, as integration is essential for optimizing the performance of a speech understanding system. In this section, we describe our efforts in integrating speech and syntax, and use this as the basis for incorporating other natural language components. Before describing the algorithms for integration, we will first review the syntactic component of our natural language system (for a detailed discussion, see Chapter 1). As described previously, our system uses a unification grammar for representing the syntax. A unification grammar is essentially a context-free grammar (CFG) augmented with variables. The algorithm used for performing syntactic analysis operates word-synchronously, left-to-right and bottom-up and computes all possible parses of the input. It is similar to the CKY parsing algorithm that appears in the literature on context-free grammars. It starts at the terminals and iteratively derives larger grammatical constituents spanning the smaller ones that have already been found. Building a larger constituent from subconstituents involves *unification* (see Chapter 1)—the process of matching terms (made of complex expressions) in the grammar, requiring recursive computation, which is a compute-intensive and memory-intensive process. This algorithm is shown to have a computational complexity proportional to N^3, where N is the length of the input text.

We describe two parsing algorithms that have been implemented for integrating speech and syntax. Both are natural extensions of the text parser. The first is a time-synchronous parsing algorithm that operates at the resolution of the frame (10 millisecond). However, this algorithm is extremely complex computationally, rendering it impractical. Our second implementation is a word-synchronous parser more similar to the text parser. It takes advantage of the redundancy across time frames by combining similar constituents that occur across different time intervals into a single constituent and parsing with only this single contituent. A computational saving of two orders of magnitude has been realized using this algorithm.

We describe the two speech parsers below. In subsequent sections, all discussions pertain only to the word-synchronous parser, which is our current implementation.

3.2.1 The Time-Synchronous Speech Parser

Figure 3-3 presents the algorithm for the time-synchronous lattice parser.

As can be seen, this lattice parser is similar in many respects to the text parser. (Compare it to the algorithm in Figure 1-1.) The parser builds the table `dr[i,j]` starting from time j=1 and marches left to right, filling the table with valid grammatical constituents. What is distinctly different is that i and j range over time/frame positions within the utterance rather than over word positions, and that each grammatical constituent has been augumented with an acoustic likelihood score `S[i,j]`. The process of parsing involves matching terms to derive larger constituents as well as combining the acoustic scores from substituents to arrive at a new acoustic score. The major drawback of this algorithm is its computation and storage complexity. Since the parsing algorithm runs in time proportional to the length of the input, and the length in this case is T, the number of frames in the speech utterance,

```
;;; First compute the word lattice
   for all terminals W
       Compute acoustic likelihood score S(i,k|W), i < k < T
       using DTW

;;; For all ending time
for k = 1 to T

   ;;; For all starting times
   for i = k-1 to 0 by -1

       ;;; Compute chart entries for time interval <i,k>
       dr[i,k] =

           {(A → W. ε, S[i,k|W]) | W ∈ input [i,k]} ∪

              {(A → α B. β, S[i,j|A] + S[j,k|B] |

                    (A → α. B β , S[i,j|A]) ∈ dr[i,j]

                    & (B → γ., S[j,k|B]) ∈ dr[j,k]}


       MaxScore[i,k|A] = Max S[i,k|A]   for all (A → δ.) ∈ dr[i,k]


       Traceback[i,k|A] = (A → δ.) = Arg(MaxScore[i,k|A])


       i < j < k
```

where

`dr[i, j]`	= dotted rule table containing of grammatical constituents spanning time interval i,j $i<j$
A, B	= lefthand sides of rules in the grammar
$\alpha, \beta, \gamma, \delta, \varepsilon$	= symbols deriving arbitrary number of terminals
`MaxScore[i,k,A]`	= the best scoring grammatical constituent spanning `input[i,k]` with **A** as the lefthand side
`Traceback[i,k|A]`	= the lefthand side of **A** → δ. with the best score for `input[i,k]`

Figure 3-3: Time-synchronous Lattice Parsing Algorithm

this algorithm would run in time proportional to 300^3 (assuming an utterance is 3 seconds long)! Also, as stated, the algorithm only keeps the single best scoring parse for a time interval <i,j> and throws away all others. Alternate syntactical interpretation of the same input are explicitly discarded—making subsequent application of semantics impossible. We propose a superior parsing algorithm—the word-synchronous parser—described below.

3.2.2 The Word-Synchronous Speech Parser

In the time-synchronous parser, the entries in `dr[i, j]` contain theories of the form:

`(A → B. C D, S[i, j])`

where B spans `input[i, j]` with acoustic score `S[i, j]`. In all likelihood, `dr[i, j+1]` would include

`(A → B. C D, S[i, j+1])`

where the same dotted rule A → B. C D is being computed twice.

As unification is expensive computationally, much could be gained by removing the redundant representations in the parse table, and thereby minimizing the number of unifications computed. One way to achieve this is by grouping these two theories into a single theory,

`(A → B. C D , {S[i, j], S[i, j+1]})`

In fact, one could collect all neighboring theories into a single theory,

`(A → B. C D , {S[i, j]})`

where `{S[i, j]}` is the set of scores, and associated with each is a starting time $i, i \in \langle Imin Imax \rangle$, and an ending time $j, j \in \langle Jmin Jmax \rangle$. In effect, this groups a contiguous region of the input utterance into word units, and then applies the parser at the word level to find the best syntactic parse. This is the word-synchronous lattice parser. Figure 3-4 presents the algorithm for the word-synchronous parser.

The operator `##` is defined as the concatenation of two sets of acoustic scores `{S[t1,t2]}` and `{S[t3,t4]}` to derive a new set of scores spanning the intervals of the two sets, using the following DP algorithm:

```
For ta ∈ <T1min,T1max>
    For tb ∈ <T4min,T4max>
        S[ta,tb] =  Max(S[ta,ti] + S[ti+1,tb])
                     ti

                    where ti ∈ <T2min,T2max> ∩ <T3min,t3max>
```

While the computational complexity of the time-synchronous algorithm was T^3, that of the word-synchronous algorithm is N^3, where N is now the estimated number of words in the speech signal; part of the task of the parser is to determine the true word sequence from the signal, and therefore the value of N. The implementation of the word-synchronous lattice parser is described in detail in Section 3.4.

```
First for all terminals W
    compute (W, {S[t1,t2]})

;;; For ending positions k (in words)
for k = 1 to N

    ;;; For starting positions i (in words)
    for i = k-1 to 0 by -1

        ;;; Compute chart entries for word positions <i,k>
        dr[i,k] =

                (if i+1 = k

                    {(A → W. α , {S[t1,t2]}) | W ∈ input [i,k]}

                 else

                    {(A → α B. β , {S[t5,t6]}) |

                     (A → α. B β , {S[t1,t2]}) ∈ dr[i,j]

                      & (B → γ. , {S[t3,t4]}) ∈ dr[j,k]}

                     i < j < k
                }

                ∪

                {(A → B. α , {S[t5,t6]}) | (B → γ. , {S[t5,t6]}) ∈ dr[i,k]}

                ∀ (A → B α) ∈ P

    where
            t1 ∈ <T1min,T1max>, t2 ∈ <T2min,T2max>
            t3 ∈ <T3min,T3max>, t4 ∈ <T4min,T4max>
            t5 ∈ <T5min,T5max>, t6 ∈ <T6min,T6max>
    and
            {S[t5,t6]} = {S[t1,t2]} ## {S[t3,t4]} ≠ {}
```

Figure 3-4: Word-synchronous Parsing Algorithm

3.3 Integrating Semantics

Our initial strategy for applying semantic interpretation to the speech parser is similar to that in the text parser, i.e., after syntactic analysis has been completed. To allow for this without incurring the cost of repeating the same parsing computation over and over again for parsed constituents that are the same but with different parse trees, a scheme for representing the entries in the parse table **dr** was devised. It is as follows. The entries in the parse table **dr[i, j]** for a particular *i* and *j* is partitioned into equivalence classes. Within each equivalence class are theories having been parsed to the same grammatical expression (Gexpr), but with possibly different parse trees corresponding to different ways of parsing a particular word sequence in the lattice as well as to those that correspond to parses of different word sequences which happened to have resulted in the same parsed constituent. Each equivalence class is headed by a representative Gexpr template on which unification matching is performed (therefore, only a single unification match is performed per class); however, tree building and speech concatenation computation (unique to each member within the class) are done separately for each member within the class.

To illustrate,

```
If (A → B. C D , {S[t1,t2]}) ∈ dr[i,j]

  & {<C>: (C → E F. {S[t3,t4]})
          (C → E' F'. {S[t3',t4']})
          (C → E'' F''. {S[t3'',t4'']})
    } ∈ dr [j,k]

    Then {<A → B C. D>: (A → B C. D {S[t5,t6]})
                        (A → B C. D {S[t5',t6']})
                        (A → B C. D {S[t5'',t6'']})
         } ∈ dr[i,k]
```

where

<C> = representative Gexpr template for the equivalence class C
<A → B C. D> = resulting representative dotted rule.

By representating the parse table in this way, the parser is able to find all parses of the input without replicating the same work, which in the worst case could have exponentiating effects on computation.

Finally, semantic interpretation is performed by first finding the set of complete constituents (i.e., all entries of the form **((START) {SCORES})** in the parse table (i.e., **dr[0, i]** $i>0$) that span the entire utterance <1,*T*>, and then by finding the best scoring one of those which is also semantically meaningful. This gives us the single best answer that is optimal with respect to speech, syntax, and semantics.

3.4 System Implementation

Many practical issues were encountered in the implementation of our SLS system on the Symbolics Lisp Machine (LISPM). In this section, a representative subset of these issues will be highlighted and methods for handling them discussed. The issues and discussions are relevant only to the current implementation—the word-synchronous lattice parser.

3.4.1 Silence Handling

The word lattice computed by the speech component contains words that are in the speech lexicon, including the word **SILENCE**, representing intervals in the utterance where the model for **SILENCE** matched well against the input speech (as silence is located at utterance beginnings, utterance endings, at the beginning of plosives, actual pauses in speech, etc). Since silence, or pause, is not explicitly handled in the natural language grammar (in the future, we may include pause detections to help identify phrase boundaries), something must be done to eliminate these silences in the word lattice while maintaining its integrity. We handle this by merging silences into the neighboring words: all words that have silences as neighbors would also have another instance created that has its boundary extended to include silence, with the proper acoustic scores included. By incorporating this silence merging stage as a preprocess to the parser, we have eliminated the need to modify the grammar/parser so it could handle silence explicitly, while ensuring that all of the input signal is accounted for.

3.4.2 Search Strategies

Search strategy design is by far the most important task in designing and implementing a system dealing with a large search space such as the one we are building now, as efficient search strategies can mean orders of magnitude reduction in both computation and memory requirements. In this section we describe some of the methods employed in our system that are used to prune down the search space.

3.4.2.1 Conditions for Search Termination

As mentioned previously, the computational complexity of the word-synchronous parser is proportional to N^3, where N is the estimated/computed length of an utterance in words. To minimize unnecessary computation, one needs to detect N as early as possible and terminate search (the bottom up parser could go on for a long time beyond the actual number of words in the input). The algorithm that we have devised is as follows: before each parse, we compute the best scoring word sequence from the word lattice without using any grammar constraints (i.e., all words can follow any word). The resulting acoustic score of this answer is used as an upper bound on the score of the best scoring word sequence allowed by the grammar. The condition for terminating search is satisfied when we are at some position k in the parsing algorithm where a complete grammatical constituent (with the symbol **(START)**) spanning the entire utterance is found in `dr[0,k-1]` with a score within a threshold of the upper bound score. The

reason for choosing to look at `dr[0,k-1]` rather than `dr[0,k]` is a subtle but well-motivated one: a valid complete constituent may have short function words (such as "a" or "the") deleted from it and still score well enough to satisfy the search termination criterion. In other words, we always want to compute one more word given that we think that we have found the "correct" word sequence. By hedging against single word deletions, we are also relying on the assumption that more word deletions will deteriorate the overall score to the level where the threshold test (against the upper bound score) can no longer be satisfied. Finally, the best answer is computed by searching over all theories `((START) (SCORES))` in `dr[0,i]`, $i>0$ that span the utterance from time 1 to T and finding the theory with the best score.

3.4.2.2 Reduction In Time Resolution

A simple scheme to reduce the computation is to down-sample the backward DTW in computing the word lattice to compute at every T frame (for example, skip every other frame). This would reduce the speech lattice computation by the same factor, and would reduce the score concatenation operation (##) in the parsing by the same factor squared. This is a simple and straightforward (and well understood) method for cutting down on computational load with minimal loss in performance, and we, in fact, make this the default mode of operation for our system. Currently, we use a time resolution of two (skip every other frame) in running our system.

3.4.2.3 Word Lattice Pruning

As described earlier, the speech component computes a very dense word lattice which potentially would include all words in the lexicon with scores between every time interval $<i,j>$. The motivation behind using a such a dense word lattice in the parser is to be sure with probability close to one that the correct words would be included at the right place in the lattice. However, to consider such a lattice in its entirety would not only be a waste of effort since most of words are in fact just noise (as our models of the word are probabilistic), but would be computationally impractical for our bottom-up parser. An integral part of designing the lattice parsing algorithm is to come up with ways of reducing the size of the lattice and yet ensuring that the correct word sequence is present.

A straightforward approach is to prune down the word lattice, keeping only those words that score reasonably well (all those that have an average score per frame > δ, where δ is a predetermined threshold) and ignoring all those that scored poorly acoustically. The key, then, is to determine δ so that it would result in a high hit rate for the correct words while minimizing the detection of irrelevant words. In practice, we found it very difficult if not impossible to find such a δ that would fullfill these two conflicting requirements, as words have widely varying acoustic scores not only among themselves, but across contexts and environments. In fact, one can imagine that in the worst case, a distinct threshold would be needed for every word in every context. Fortunately, from empirical evidence as well as from our knowledge of acoustics, we infer that the average acoustic scores of words in general tend to fall into two broad classes: short words (most function words, such as "a", "the", "of", that have two syllables or less), and long words (such as proper nouns like "Frederick" and "Westpac"). The short words, since they are short in duration and are often reduced in spoken utterances, tend to score poorly acoustically; whereas the long words are often spoken more carefully, and would have consistently higher acoustic scores. This suggested the use of dual thresholds—one for short words and the other for long words. In fact, we have generalized this to an

arbitrary number of classes, with short words on one end of the spectrum and long words on the other, with other word classes in between, with the guiding principle that words with fewer syllables would tend to be more variable acoustically, and therefore would need smaller thresholds, and vice versa. Currently, our system employs four levels of word level pruning. And in practice, we have found that this multi-level pruning reduces the size of the lattice (as measured by the number of words, each with a set of boundaries with associated acoustic scores) by at least a factor of two. This reduces the parsing computation dramatically as the lattice size has an exponentiating effect on the size of the parse table.

3.4.2.4 Pruning During Speech Parsing

One strategy for pruning the search space during parsing is the time-synchronous beam search used in the BYBLOS speech recognition system. However, the beam search used in the BYBLOS system always compares theories spanning the same time interval from time 1 (beginning of utterance) to current time t, whereas the bottom-up parser used here generates theories that can span arbitrary time intervals $<t1,t2>$. We have modified the beam search to work for our parsing strategy as follows. During parsing, the maximum score spanning a particular time interval $<t1,t2>$ (for all time intervals) is computed, and theories that span $<t1,t2>$ are only kept if their score is within a threshold (the beam) of the maximum; all others are eliminated. This method is less effective when theories are short (spanning short time intervals), and much more effective as theories become long in duration. Empirically, we've found this pruning strategy reduces computation significantly over no pruning.

3.5 Current Status and Future Work

Currently, we have an integrated system that runs on actual speech utterances; the speech component produces a word-lattice, on which the natural language component performs parsing to find the most likely interpretation of the input utterance. However, a critical problem remains to be solved. On input utterances that are long, or are particularly ambiguous acoustically, producing a large word lattice (and this happens quite often), the system runs into severe search problems, requiring prohibitively large amounts of computation and memory. On such occasions, the machine simply runs out of virtual memory and crashes. The solution to this problem is to come up with methods and algorithms to significantly cut down on the search space. Some ideas to try include more efficient pruning of the word-lattice, top-down prediction, and semantic filtering. For word lattice pruning, one can imagine a pruning strategy that is data driven, rather than using thresholds that are fixed apriori. In prediction, we want to use top-down information (whereas our parser operates strictly bottom-up) to reduce the size of the parse table, and therefore minimize computation and storage. Likewise, semantics can be used incrementally (at the constituent level) to filter out semantically anomalous syntactic parses in the parse table. We will incorporate one or more of these methods and test their effectiveness within the context of the overall search strategy design.

References

[1] Alexander, D. and W.J. Kunz.
Some Classes of Verbs in English.
distributed by Indiana University Linguistics Club. Bloomington, Indiana, 1964.
Linguistics Research Project, Indiana University, F. W. Householder, Jr., Principal Investigator.

[2] Bresnan, Joan.
Contraction and the Transformational Cycle in English.
distributed by the Indiana University Linguistics Club. Bloomington, Indiana, 1978.
Originally written in 1971.

[3] Bridgeman, Loraine I., Dale Dillinger, Constance Higgins, P. David Seaman, Floyd A. Shank.
More Classes of Verbs.
distributed by Indiana University Linguistics Club. Bloomington, Indiana, 1965.
Linguistics Research Project, Indiana University, F. W. Householder, Jr., Principal Investigator.

[4] W.J.H.J. Bronnenberg, H.C. Bunt, S.P.J. Landsbergen, R.J.H. Scha, W.J. Schoenmakers and E.P.C. van Utteren.
The Question Answering System PHLIQA1.
In L. Bolc (editor), *Natural Language Question Answering Systems*. Macmillan, 1980.

[5] Chomsky, Noam.
Aspects of the Theory of Syntax.
MIT Press, Cambridge, Massachusetts, 1965.

[6] Y.L. Chow, M.O. Dunham, O.A. Kimball, M.A. Krasner, G.F. Kubala, J. Makhoul, P.J. Price, S. Roucos, and R.M. Schwartz
BYBLOS: The BBN Continuous Speech Recognition System.
In *International Conference on Acoustics, Speech, and Signal Processing*, pages 89-93. IEEE, Dallas, Texas, April, 1987.

[7] DeBruin, Jos and Scha, Remko J H.
The Interpretation of Relational Nouns.
In *Proceedings of the ACL*. Association for Computational Linguistics, June, 1988.
To Appear.

[8] Gazdar, Gerald, Ewan Klein, Geoffrey Pullum, Ivan Sag.
Generalized Phrase Structure Grammar.
Harvard University Press, Cambridge, Massachusetts, 1985.

[9] Graham, Susan L., Michael A. Harrison, and Walter L. Ruzzo.
An Improved Context-free Recognizer.
ACM Transactions on Programming Languages and Systems 2(3):415-461, 1980.

[10] Haas, Andrew.
Parallel Parsing for Unification Grammar.
In *Proceedings of the Tenth International Joint Conference on Artificial Intelligence*, pages 615-618. IJCAI, Milan, Italy, August, 1987.

[11] Haas, Andrew.
A Parsing Algorithm for Unification Grammar.
forthcoming.

[12] Harman, Gilbert.
Generative Grammars without Transformation Rules: A Defense of Phrase Structure.
Language 39:597-616, 1963.

[13] Heidorn, George E.
English as a Very High Level Language for Simulation.
SIGPLAN Notices 9(4):91-100, April, 1974.

[14] Heidorn, George E.
Automatic programming through natural dialogue: a survey.
IBM Journal of Research and Development 20(4):302-313, July, 1976.

[15] Ingria, Robert J.
A Summary of Verb Complement Types in English.
forthcoming.

[16] Ingria, Robert J.
Features in the BBN ACFG for Selected Verbs from the COBUILD Corpus.
forthcoming.

[17] Frederick Jelinek.
Continuous Speech Recognition by Statistical Methods.
Proceedings of the IEEE 64(4):532-556, April, 1976.

[18] Montague, R.
The Proper Treatment of Quantification in Ordinary English.
In J. Hintakka, J.Moravcsik and P.Suppes (editors), *Approaches to Natural Language. Proceedings of the 1970 Stanford Workship on Grammar and Semantics*, pages 221-242. Dordrecht: D.Reidel, 1973.

[19] Pereira, Fernando.
Extraposition Grammars.
American Journal of Computational Linguistics 7(4):243-256, 1981.

[20] Procter, Paul et al, eds.
Longman Dictionary of Contemporary English.
Longman Group Limited, Harlow and London, 1978.

[21] Scha, Remko J.H.
Distributive, Collective and Cumulative Quantification.
In Jeroen Groenendijk, Theo M.V. Janssen, Martin Stokhof (editors), *Formal Methods in the Study of Language. Part 2*, pages 483-512. Mathematisch Centrum, Amsterdam, 1981.

[22] Scha, Remko J.H.
Logical Foundations for Question-Answering.
Philips Research Laboratories, Eindhoven, The Netherlands, 1983.
M.S.12.331.

[23] Scha, Remko J.H. and Stallard, David G.
Multi-level Plurals and Distributivity.
In *Proceedings of the ACL.* Association for Computational Linguistics, June, 1988.
To Appear.

[24] R. Schwartz, Y. Chow, O. Kimball, S. Roucos, M. Krasner, J. Makhoul.
Context-Dependent Modeling for Acoustic-Phonetic Recognition of Continuous Speech.
In *International Conference on Acoustics, Speech, and Signal Processing*, pages 1205-1208. IEEE, Tampa, Florida, March, 1985.

[25] Shieber, Stuart M.
An Introduction to Unification-Based Approaches to Grammar.
CSLI (Center for the Study of Language and Information), Stanford University, Stanford, CA, 1986.

[26] Tarski, Alfred.
Der Wahrheitsbegriff in den Formalisierten Sprachen.
Studia Philosophica 1 :261-405, 1935.
translated as "The concept of truth in formalized languages," *Logic, Semantics, and Mathematics*, pp. 152-278, ed. by A. Tarski, Oxford: Clarendon Press.

[27] Visser, F. Th.
An Historical Syntax of the English Language; Part One: Syntactical Units with One Verb.
E. J. Brill, Leiden, 1963.

[28] Visser, F. Th.
An Historical Syntax of the English Language; Part Two: Syntactical Units with One Verb (Continued).
E. J. Brill, Leiden, 1966.

[29] Visser, F. Th.
An Historical Syntax of the English Language; Part Three, First Half: Syntactical Units with Two Verbs.
E. J. Brill, Leiden, 1969.

[30] Visser, F. Th.
An Historical Syntax of the English Language; Part Three, Second Half: Syntactical Units with Two Verbs and with More Verbs.
E. J. Brill, Leiden, 1973.

Appendix A
A Parsing Algorithm for Unification Grammar
Andrew Haas

Abstract

We describe a table-driven parser for unification grammar that combines bottom-up construction of phrases with top-down filtering. This algorithm works on a class of grammars called depth-bounded grammars, and it is guaranteed to halt for any input string. Unlike many unification parsers our algorithm works directly on a unification grammar—it does not require that we divide the grammar into a context-free "backbone" and a set of feature agreement constraints. We give a detailed proof of correctness. For the case of a pure bottom-up parser, our proof does not rely on the details of unification—it works for any pattern matching technique that satisfies certain simple conditions.

Unification grammars have the formal power of a Turing machine. Thus there is no algorithm that finds all parses of a given sentence in any unification grammar and always halts. Some unification grammar systems just live with this problem. The usual parser for definite clause grammar may enter an infinite loop if the grammar contains left recursion, and it is the task of the grammar writer to avoid this. Generalized phrase structure grammar avoids this problem because it has only the formal power of context-free grammar, but according to Shieber (1985) this is not adequate for describing human language.

Lexical functional grammar employs a better solution. A lexical functional grammar must include a finitely ambiguous context-free grammar, which we will call the context-free backbone (Barton , 87, p. 105). A parser for lexical functional grammar first builds the finite set of context-free parses of the input and then eliminates those that don't meet the other requirements of the grammar. This method guarantees that the parser will halt.

This solution may be adequate for lexical functional grammars, but for other unification grammars finding a finitely ambiguous context-free backbone is a problem. Suppose we use the notation of definite clause grammar. An obvious way to build a context-free backbone is to keep only the topmost function letters in each rule. Thus the rule

(s) → (np :p :n) (vp :p :n)

becomes

s → np vp

Suppose we use a simple X-bar theory. Let (major-category :type :bar-level) denote a phrase in a major category. A noun phrase may consist of a single noun, for instance "John". This suggests a rule like this:

(major-category (n) 2) → (major-category (n) 1)

In the context-free backbone this becomes

major-category → major-category

so the context-free backbone is infinitely ambiguous. One could devise more elaborate examples, but this one suffices to make the point: not every natural unification grammar has an obvious context-free backbone. Therefore we need a parser that does not require us to find a context-free backbone, but works directly on a unification grammar.

We propose to guarantee that the parsing problem is solvable by restricting ourselves to depth-bounded grammars. A unification grammar is depth-bounded if for every L > 0 there is a D > 0 such that every parse tree for a string of L symbols has depth less than D. In other words, the depth of a tree is bounded by the length of the string it derives. A context-free grammar is depth-bounded if and only if every string of symbols is finitely ambiguous, but for unification grammars this is false: depth-boundedness is a stronger property than finite ambiguity.

Depth-bounded unification grammars have more formal power than context-free grammars. As an example we give a depth-bounded grammar for the language xx, which is not context-free. Suppose the terminal symbols are A through Z. We introduce function letters A' through Z' to represent the terminals. The rules of the grammar are as follows, with # denoting the empty string.

```
(s) → (x :1)(x :1)
(x (cons :a :1)) → (pre-terminal :a) (x :1)
(x (nil)) → #
(pre-terminal (A')) → A
...
(pre-terminal (Z')) → Z
```

The reasoning behind the grammar should be clear—(x (cons (A') (cons (B') (nil)))) derives AB, and the first rule guarantees that every sentence has the form xx. The grammar is depth-bounded because the depth of a tree is a linear function of the length of the string it derives. A similar grammar can derive the crossed serial dependencies of Swiss German, which according to Shieber (1985) no context-free grammar can derive. It is clear where the extra formal power comes from: a context-free grammar has a finite set of non-terminals, but a unification grammar can build arbitrarily large non-terminal symbols.

It remains to show that there is a parsing algorithm for depth-bounded unification grammars. We have developed such an algorithm, based on the context-free parser of Graham, Harrison and Ruzzo (1980), which is a table-driven parser. If we generalize the table-building algorithm to a unification grammar in an obvious way, we get an algorithm that is guaranteed to halt for all depth-bounded grammars (not for all unification grammars). Given that the tables can be built, it is easy to show that the parser halts on every input. If the grammar is not depth-bounded the table-building algorithm will enter an infinite loop, and it is up to the grammar writer to fix this. In practice we have not found this troublesome. In any case it is better than having a parser (such as the usual definite clause grammar parser) that may parse a hundred sentences and then enter an infinite loop on the hundred and first.

Sections A.1 and A.2 of this paper defines the basic concepts of our formalism. Section A.3 proves the

soundness and completeness of our simplest parser, which is purely bottom-up and excludes rules with empty right hand sides. Section A.4 admits rules with empty right sides, and Section A.5 adds top-down filtering.

A.1 Basic Concepts

A *typed language* is five-tuple $L = (T,F,V,s_1,s_2)$, where T is a finite set of types, F is a finite set of function letters, V is a countably infinite set of variables, s_1 is a function from $F \cup V$ onto T, and s_2 is a function from F into T^*. The function s_1 assigns a type to each function letter and variable. The function s_2 assigns types to the argument positions of each function letter. We assume that T, F and V are pairwise disjoint.

A *term* is either a variable or an expression $(f\ x_1...x_n)$, where f is an n-adic function letter and $x_1...x_n$ are terms. The type of a variable v is $(s_1\ v)$, and the type of a term $(f\ x_1...x_n)$ is $(s_1\ f)$. A term is well-typed if it is a variable, or if it has the form $(f\ x_1...x_n)$ where $x_1...\ x_n$ are well-formed terms, $t_1...t_n$ are the types of $x_1...\ x_n$ respectively, and $(s_1\ f) = t_1...t_n$. From now one we shall consider only well-typed terms. A ground instance of a term t is a substitution instance of t that contains no variables and has the same type as t. In other words, it is a term fomred by replacing each variable v of t with a term of the same type as v that contains no variables.

A *unification grammar* is a four-tuple G = (L, Q, P, S) where L is a typed language and Q is a set of terminal symbols. Q is disjoint from the sets of variables and function letters in L. P is a finite set of rules; each rule has the form $(A \rightarrow \alpha)$, where A is a term of L and α is a sequence of terms of L and symbols from Q. S is a ground term of L (that is, a term without variables). S is called the start symbol of G.

The ground grammar for G is the 4-tuple (N, Q, P', S), where N is the set of ground terms of L, Q is the set of terminals of G, P' is the set of all ground instances of rules in P, and S is the start symbol of G. If N and P' are finite the ground grammar is a context-free grammar. If N or P is infinite the ground grammar is not a context-free grammar, and it may generate a language that is not context-free. Nonetheless we can define derivation trees just as in a cfg. A derivation tree is an A-tree if the non-terminal A labels its root. The yield of a derivation tree is the string formed by reading the symbols at its leaves from left to right. As in a cfg, $A \Rightarrow \alpha$ iff there is an A-tree with yield α. The language generated by a ground grammar is the set of terminal strings derived from the start symbol. The language generated by a unification grammar is the language generated by its ground grammar.

Suppose t_1 and t_2 are types, and there is a function letter of type t_1 that has an argument of type t_2. Then we say that $t_1 > t_2$. If the relation > is a partial order and D is the number of types, every term of the ground grammar has depth $\leq D$. Since there are only a finite number of function letters in the language L, each taking a fixed number of arguments, the number of possible terms of depth D is finite. Then the ground grammar is finite.

A ground grammar G' is depth-bounded if for every integer n there exists an integer d such that every derivation tree in G' with a yield of length n has a depth less than d. In other words, a depth-bounded grammar cannnot build an unbounded amount of tree structure from a bounded number of symbols. A unification grammar G is depth-bounded if its ground grammar is depth-bounded.

We have defined the semantics of our grammar formalism without mentioning unification. This is deliberate: for us unification is a computational tool, not a part of the formalism. It might be better to call the formalism "substitution grammar", but the other name is already established.

Notation: The letters A, B, C denote symbols of a ground grammar, including terminals and non-terminals. Lower-case Greek letters denote strings of symbols. $\alpha[i\ k]$ is the substring of α from space i to space k, where the space before the first symbol is space zero. # is the empty string. We write $x \cup y$ or $(\cup\ x\ y)$ for the union of sets x and y, and also $(\cup\ i < j < k\ (f\ j))$ for the union of the sets $(f\ j)$ for all j such that $i < j < k$.

If α is the yield of a tree t, then to every occurrence of a symbol A in α there corresponds a leaf of t labeled with A. To every node in t there corresponds an occurrence of a substring in α—the substring dominated by that node. Here is a lemma about trees and their yields that will be useful when we consider top-down filtering.

Lemma 2.6. Suppose t is a tree with yield $\alpha\beta\alpha'$ and n is the node of t corresponding to the occurence of β after α in $\alpha\beta\alpha'$. Let A be the label of n. If t' is the tree formed by deleting all descendants of n from t, the yield of t' is $\alpha A\alpha'$

Proof: This is intuitively clear, but the careful reader may prove it by induction on the depth of t.

A.2 Operations on Sets of Rules and Terms

The parser must find the set of ground terms that derive the input string and check whether the start symbol is one of them. We have taken the rules of a unification grammar as an abbreviation for the set of all their ground instances. In the same way, the parser will use sets of terms and rules containing variables as a representation for sets of ground terms and ground rules. In this section we show how various functions needed for parsing can be computed using this representation.

A grammatical expression, or g-expression, is either a term of L, the special symbol Nil, or a pair of g-expressions. The letters u, v, w, x, y, and z denote g-expressions, and X, Y and Z denote sets of g-expressions. We use the usual LISP functions and predicates to describe g-expressions. [x y] is another notation for (cons x y). For any substitution s, $(s\ (\text{cons}\ x\ y)) = (\text{cons}\ (s\ x)\ (s\ y))$ and $(s\ \text{Nil}) = \text{Nil}$. A selector is a function from g-expressions to g-expressions formed by composition from the functions Car, Cdr, and Identity. Thus a selector picks out a sub-expression from a g-expression. A constructor is a function that maps two g-expressions to a g-expression, formed by composition from the functions Cons, Car, Cdr, Nil, $(\lambda\ x\ y.\ x)$, and $(\lambda\ x\ y.\ y)$. A constructor builds a new g-expression from parts of two given g-expressions. A g-predicate is a function from g-expressions to Booleans formed by composition from the basic functions Car, Cdr, $(\lambda\ x.\ x)$, ConsP, and Null.

Let (ground X) be the set of ground instances of g-expressions in X. If f is a selector function, let (f X) be the set of all (f x) such that $x \in X$. If p is a g-predicate, let (separate p X) be the set of all $x \in X$ such that (p x). The following lemmas are easily established from the definition of (f x) for a g-expression x.

Lemma 2.1. If f is a selector function, (f (ground X)) = (ground (f X)).

Lemma 2.2. If p is a g-predicate , (separate p (ground X)) = (ground (separate p X)).

Lemma 2.3. (ground X ∪ Y) = (ground X) ∪ (ground Y).

Lemma 2.4. If x is a ground term, x ∈ (ground X) iff x is an instance of some y ∈ X.

Lemma 2.5. (ground X) is empty iff X is empty.

These lemmas tell us that if we use sets X and Y of terms to represent the sets (ground X) and (ground Y) of ground terms, we can easily construct representations for (f (ground X)), (separate p (ground X)), and (ground X) ∪ (ground Y). Also we can decide whether a given ground term is contained in ground X) and whether (ground X) is empty. All these operations will be needed in the parser.

The parser requires one more type of operation, as follows.

Definition. Let f_1 and f_2 be selectors and g a constructor, and suppose (g x y) is well-defined whenever (f_1 x) and (f_2 y) are well-defined. The *symbolic product* defined by f_1, f_2, and g is the function

$$(\lambda X\ Y.\ \{\ (g\ x\ y) \mid x \in X \wedge y \in Y \wedge (f_1\ x) = (f_2\ y)\ \})$$

where X and Y range over sets of ground g-expressions. Note that (f_1 x) = (f_2 y) is considered false if either side of the equation is undefined.

If X is a set of g-expressions and n an integer, (rename X n) is an alphabetic variant of X. For all X, Y, m, and n, if m ≠ n then (rename X n) and (rename Y m) have no variables in common. The following theorem tells us that if we use sets of terms X and Y to represent the sets (ground X) and (ground Y) of ground terms, we can use unification to compute any symbolic product of (ground X) and (ground Y). We assume the basic facts about unification as in Robinson (1965).

Theorem 2.1. If h is the symbolic product defined by f_1, f_2, and g, and X and Y are sets of g-expressions, then

```
(h (ground X) (ground Y)) =
(ground {(s (g u v)) | u ∈ (rename X 1) ∧ v ∈ (rename Y 2)
                     ∧ s is the m.g.u. of (f1 u) and (f2 v)}
```

Proof: The first step is to show that if Z and W share no variables

```
(1)
{(g z w) | z ∈ (ground Z) ∧ w ∈ (ground W) ∧ (f1 z) = (f2 w)}
 =
(ground {(s (g u v)) | u ∈ Z ∧ v ∈ W
                     ∧ s is the m.g.u. of (f1 u) and (f2 v)
```

})

Consider any element of the right side of equation (1). It must be a ground instance of (s (g u v)), where u ∈ Z, v ∈ W, and s is the m.g.u. of (f_1 u) and (f_2 v). Any ground instance of (s (g u v)) can be written as (s' (s (g u v))), where s' is chosen so that (s' (s u)) and (s' (s v)) are ground terms. Then (s' (s (g u v))) = (g (s' (s u)) (s' (s v))) and (f_1 (s' (s u))) = (s' (s (f_1 u))) = (s' (s (f_2 v))) = (f_2 (s' (s v))). Therefore (s' (s (g u v))) belongs t the set on the left side of equation (1).

Next consider any element of the left side of (1). It must have the form (g z w), where z ∈ (ground Z), w ∈ (ground W), and (f_1 z) = (f_2 w). Then for some u ∈ Z and v ∈ W, z is a ground instance of u and w is a ground instance of v. Since u and v share no variables, there is a substitution s' such that (s' u) = z and (s' v) = w. Then (s' (f_1 u)) = (f_1 (s' u)) = (f_2 (s' v)) = (s' (f_2 v)), so there exists a most general unifier s for (f_1 u) and (f_2 v), and s' is the composition of s and some substitution s''. Then (g z w) = (g (s'' (s u)) (s'' (s v))) = (s'' (s (g u v))). (g z w) is a ground term because z and w are ground terms, so (g z w) is a ground instance of (s (g u v)) and therefore belongs to the set on the right side of equation (1).

We have proved that if Z and W share no variables,

```
(2)  (h (ground Z) (ground W)) =
     (ground {(s (g u v)) | u ∈ Z ∧ v ∈ W
                          ∧ s is the m.g.u. of (f1 u) and (f2 v)}
```

For any X and Y, (rename X 1) and (rename Y 2) share no variables. Then we can let Z = (rename X 1) and W = (rename Y 2) in formula (2). Since (h (ground X) (ground Y)) = (h (ground (rename X 1)) (ground (rename Y 2))), the theorem follows by transitivity of equality. This completes the proof.

Definition. Let f be a function from sets of g-expressions to sets of g-expressions, and suppose that when X ⊆ X' and Y ⊆ Y', (f X Y) ⊆ (f X' Y'). Then f is *monotonic*.

All symbolic products are monotonic functions, as the reader can easily show from the definition of symbolic products. Indeed every function in the parser that returns a set of g-expressions is monotonic.

A.3 The Parser without Empty Symbols

Our first parser does not allow rules with empty right sides, since these create complications that obscure the main ideas. Therefore throughout this section let G = <L, T, P, S> be a ground grammar in which no rule has an empty side. When we say that α derives β we mean that α derives β in G. Thus α ⇒ # iff α = #.

A dotted rule in G is a rule of G with the right side divided into two parts by a dot. DR is the set of all dotted rules in G. A dotted rule (A → α.β) derives a string if α derives that string. In order to compute symbolic products on sets of rules or dotted rules, we must represent them as g-expressions. We represent the rule (A → B C) as the list (A B C), and the dotted rule (A → B.C) as the pair [(A B C) (C)].

We write $A \stackrel{+}{\Rightarrow} B$ if A derives B by a tree with more than one node. The parser relies on a chain table—a table of all pairs [A B] such that $A \stackrel{+}{\Rightarrow} B$. Let C_d be the set of all [A B] such that $A \stackrel{+}{\Rightarrow} B$ by a derivation tree of depth d. Clearly C_1 is the set of all [A B] such that $(A \rightarrow B)$ is a rule of G. If S_1 and S_2 are sets of pairs of terms, define

$$(\text{link } S_1\ S_2) = \{[A\ C] \mid (\exists\ B.\ [A\ B] \in S_1 \wedge [B\ C] \in S_2)\}$$

The function "link" is equal to the symbolic product defined by f_1 = Cdr, f_2 = Car, and g = (λ x y . (cons (car x) (cdr y))). Therefore we can compute (link S_1 S_2) by applying Theorem 2.1. Clearly C_{d+1} = (link C_d C_1). Since the grammar is depth-bounded there exists a number D such that every derivation tree whose yield contains exactly one symbol has depth less than D. Then C_D is empty. The algorithm for building the chain table is this: compute C_n for increasing values of n until C_n is empty. Then the union of all the C_n's is the chain table.

Definitions. ChainTable is the set of all [A B] such that $A \stackrel{+}{\Rightarrow} B$. If S is a set of dotted pairs of symbols and S' a set of symbols, (ChainUp S S') is the set of symbols A such that $[A\ B] \in S$ for some $B \in S'$. "ChainUp" is clearly a symbolic product. If S is a set of symbols, (close S) is the union of S and (ChainUp ChainTable S)). By the definition of ChainTable, (close S) is the set of symbols that derive a symbol of S.

Let α be an input string of length $L > 0$. For each α[i k] the parser will construct the set of dotted rules that derive α[i k]. The start symbol appears on the left side of one of these rules iff α[i k] is a sentence of G. By lemma 2.4 this can be tested, so we have a recognizer for the language generated by G. With a small modification the algorithm can find the set of derivation trees of α. We omit details and speak of the algorithm as a "parser" when strictly speaking it is a recognizer only.

The dotted rules that derive α[i k] can be partitioned into two sets: rules with many symbols before the dot and rules with exactly one. The algorithm will construct the first set recursively and then construct the second set from the first. Their union is the desired set of dotted rules. Note that a dotted rule derives α[i k] with more than one symbol before the dot iff it can be written in the form $(A \rightarrow \beta B.\beta')$ where $\beta \stackrel{+}{\Rightarrow} \alpha[i\ j]$, $B \stackrel{+}{\Rightarrow} \alpha[j\ k]$, and $0 < j < k$ (this follows because $\beta \stackrel{+}{\Rightarrow} \#$ iff $\beta = \#$).

Definition. If S is a set of dotted rules and S' a set of symbols, (AdvanceDot S S') is the set of rules $(A \rightarrow \alpha B.\beta)$ such that $(A \rightarrow \alpha.B\beta) \in S \wedge B \in S'$. Clearly "AdvanceDot" is a symbolic product.

Lemma 3.1 For $i < j < k$, let (S i j) be the set of dotted rules that derive α[i j] and (S' j k) the set of symbols that derive α[j k]. The set of dotted rules that derive α[i k] with many symbols before the dot is

$$\bigcup_{i<j<k} (\text{AdvanceDot } (S\ i\ j)\ (S'\ j\ k))$$

Proof: We have

$$\bigcup_{i<j<k} (\text{AdvanceDot } \{(B \rightarrow \beta.\beta_1) \in DR \mid \beta \stackrel{+}{\Rightarrow} \alpha[i\ j]\}\ \{A \mid A \stackrel{+}{\Rightarrow} \alpha[j\ k]\})$$

=

$\cup$ {(B $\rightarrow$ βA.β_2) $\in$ DR | $\beta \Rightarrow \alpha$[i j] $\wedge$ A $\Rightarrow \alpha$[j k]) by defn. of AdvanceDot
i<j<k

=

{(B $\rightarrow$ βA.β_2) $\in$ DR | ($\exists$ j. i<j<k $\wedge$ $\beta \Rightarrow \alpha$[i j] $\wedge$ A $\Rightarrow \alpha$[j k]))

As noted above, this is the set of dotted rules that derive α[i k] with more than one symbol before the dot.

Definition. If S is a set of rules, (finished S) is the set of left sides of rules in S.

Lemma 3.2. Suppose (length α) > 1 and S is the set of dotted rules that derive α with more than one symbol before the dot. The set of symbols that derive α is (close (finished S)).

Proof: Suppose first that A $\in$ (close (finished S)). Then for some B A $\Rightarrow$ B, (B $\rightarrow$ β.) is a dotted rule, and $\beta \Rightarrow \alpha$. Then A $\Rightarrow \alpha$. Suppose next that A derives α. We show by induction that if t is a derivation tree in G and A $\Rightarrow \alpha$ by t, then A $\in$ (close (finished S)). t contains more than one node because (length α) > 1, so there is a rule (A $\rightarrow A_1 \ldots A_n$) that admits the root of t. If n > 1, (A $\rightarrow A_1 \ldots A_n$.) $\in$ S and A is in (close (finished S)). If n = 1 then $A_1 \Rightarrow \alpha$ and by induction hypothesis $A_1 \in$ (close (finished S)). Since A $\Rightarrow A_1$, A $\in$ (close (finished S)).

Definitions. RuleTable is the set of dotted rules (A $\rightarrow$.α) such that (A $\rightarrow \alpha$) is in P, the set of rules of G. If S is a set of symbols, (NewRules S) is (AdvanceDot RuleTable S).

Lemma 3.3. If S is the set of symbols that derive α, the set of dotted rules that derive α with one symbol before the dot is (NewRules S).

Proof: Expanding the definitions gives (AdvanceDot {(A $\rightarrow$.β) | (A $\rightarrow \beta$) $\in$ P} {C | C $\Rightarrow \alpha$}) = {(A $\rightarrow$ C.β') | (A $\rightarrow$ Cβ') $\in$ P $\wedge$ C $\Rightarrow \alpha$}. This is the set of dotted rules that derive α with one symbol before the dot.

Let α be a string of length L. For $0 \le i < k \le L$, define

```
(dr i k) =
  (let rules1 = (∪ i<j<k (AdvanceDot (dr i j)
                                     (∪ (finished (dr j k))
                                        (if j+1=k {α[j k]} ∅))))
    (let rules2 = (NewRules (close (if i+1=k {α[i k]} (finished rules1))))
      (∪ rules1 rules2)
  ))
```

Theorem 3.1. For $0 \le i < k \le L$, (dr i k) is the set of dotted rules that derive α[i k].

Proof: By induction on the length of input[i k]. Suppose i < j < k. By induction hypothesis (dr i j) is the set of dotted rules that derive α[i j] and (finished (dr j k)) is the set of non-terminals that derive α[j k]. Clearly (if j+1=k {α[j k]} $\emptyset$) is the set of terminals that derive α[j k], so the second argument of AdvanceDot is the set of all symbols that derive α[j k]. Then by Lemma 3.1, $rules_1$ is the set of dotted rules that derive α[i k] with many symbols before the dot. If i+1=k then (close {α[i k]}) is the set of symbols that derive α[i k], and if i+1 < k then (close (finished

$rules_1$)) is the set of symbols that derive $\alpha[i\ k]$ by lemma 3.2. In either case $rules_2$ is the set of rules that derive $\alpha[i\ k]$ with one symbol before the dot, by lemma 3.3. This completes the proof.

A.4 The Parser with Empty Symbols

Throughout this section G = <L,T,P,S> is an arbitrary unification grammar, which may contain rules whose right side is empty. If there are empty rules in the grammar the parser will require a table of symbols that derive the empty string, which we also call the table of empty symbols. Let E_d be the set of symbols that derive the empty string by a derivation of depth d, and let E'_d be the set of symbols that derive the empty string by a derivation of depth d or less. Since the grammar is depth-bounded, it suffices to construct E_d for successive values of d until a $D > 0$ is found such that E_D is the empty set.

E_0 is the empty set. $A \in E_{d+1}$ iff there is a rule $(A \rightarrow B_1 ... B_n)$ such that for each i, $B_i \stackrel{*}{\Rightarrow} \#$ at depth d_i, and d is the maximum of the d_i's. In other words: $A \in E_{d+1}$ iff there is a rule $(A \rightarrow \alpha B \beta)$ such that $B \in E_d$ and every symbol of α and β is in E'_d.

Let DR be a set of dotted rules and S a set of symbols. Define

```
(AdvanceDot* DR S) = (if DR = ∅ ∅ (∪ DR (AdvanceDot* (AdvanceDot DR S) S)
```

If DR is the set of ground instances of a finite set of rules with variables, there is a finite bound on the length of the right sides of rules in DR (because the right side of a ground instance of a rule r has the same length as the right side of r). If L is the length of the right side of the longest rule in DR, then (AdvanceDot* DR S) is well-defined because the recursion stops at depth L or before. Clearly (AdvanceDot* DR S) is the set of rules $(A \rightarrow \alpha\beta.\gamma)$ such that $(A \rightarrow \alpha.\beta\gamma) \in$ DR and every symbol of β is in S.

Let

```
S1 = (AdvanceDot* RuleTable E'd)
S2 = (AdvanceDot S1 Ed)
S3 = (AdvanceDot* S2 E'd)
S4 = (finished S3)
```

S_1 is the set of dotted rules $(A \rightarrow \alpha.\beta_0)$ such that every symbol of α is in E'_d. S_2 is then the set of dotted rules $(A \rightarrow \alpha B.\beta_1)$ such that $B \in E_d$ and every symbol of α is in E'_d. Therefore S_3 is the set of dotted rules $(A \rightarrow \alpha B \beta.\beta_2)$ such that $B \in E_d$ and every symbol of α and β is in E'_d. Finally S_4 is the set of symbols A such that for some rule $(A \rightarrow \alpha B \beta)$, $B \in E_d$ and every symbol of α and β is in E'_d. Then S_4 is E_{d+1}. In this way we can construct E_d for increasing values of d until the table of empty symbols is complete.

Definitions. Let EmptyTable be the set of terms that derive the empty string. If S is a set of dotted rules, let

(SkipEmpty S) be (AdvanceDot* S EmptyTable). Note that (SkipEmpty S) is the set of dotted rules $(A \rightarrow \alpha\beta_1.\beta_2)$ such that $(A \rightarrow \alpha.\beta_1\beta_2) \in S$ and $\beta_1 \stackrel{*}{\Rightarrow} \#$.

Let C_d be the set of pairs [A B] such that $A \stackrel{+}{\Rightarrow} B$ by a derivation tree in which the unique leaf labelled B is at depth d (note this does not imply that the tree is of depth d). C_1 is the set of pairs [A B] such that $(A \rightarrow \alpha B\beta)$ is a rule and every symbol of α and β derives the empty string. C_1 is easily computed using SkipEmpty. Also C_{d+1} = (link C_d C_1), so we can construct the chain table as before.

The parser of Section A.3 relied on the distinction between dotted rules with one and many symbols before the dot. If empty symbols are present, we need a slightly more complex distinction. We say that the string α derives β using one symbol if there is a derivation of β from α in which exactly one symbol of α derives a non-empty string. We say that α derives β using many symbols if there is a derivation of β from α in which more than one symbol of α derives a non-empty string. If a string α derives a string β, then α derives β using one symbol, or α derives β using many symbols, or both. We say that a dotted rule derives β using one (or many) symbols if the string before the dot derives β using one (or many) symbols. Note that a dotted rule derives α[i k] using many symbols iff it can be written as $(A \rightarrow \beta B\beta'.\beta_1)$ where $\beta \stackrel{*}{\Rightarrow} \alpha$[i j], $B \stackrel{+}{\Rightarrow} \alpha$[j k], $\beta' \stackrel{*}{\Rightarrow} \#$, and $i < j < k$. This is true because whenever a dotted rule derives a string using many symbols, there must be a last symbol before the dot that derives a non-empty string. Let B be that symbol; it is followed by a β' that derives the empty string, and preceded by a β that must contain at least one more symbol deriving a non-empty string.

We prove lemmas analogous to 3.1, 3.2, and 3.3.

Lemma 4.1 For $i < j < k$ let (S i j) be the set of dotted rules that derive α[i j] and (S' j k) the set of symbols that derive α[j k]. The set of dotted rules that derive α[i k] using many symbols is

$$\mathbf{(SkipEmpty} \cup \mathbf{(AdvanceDot\ (S\ i\ j)\ (S'\ j\ k)))} \quad \mathbf{i<j<k}$$

Proof: Expanding definitions and using the argument of Lemma 3.3 we have

$$\mathbf{(SkipEmpty} \cup \mathbf{(AdvanceDot\ ((B} \rightarrow \beta.\beta_1) \in \mathbf{DR} \mid \beta \stackrel{*}{\Rightarrow} \alpha\mathbf{[i\ j]\}} \quad \mathbf{i<j<k} \qquad \mathbf{(A \mid A} \stackrel{+}{\Rightarrow} \alpha\mathbf{[j\ k]}))$$

$$) =$$

$$\mathbf{(SkipEmpty\ \{(B} \rightarrow \beta\mathbf{A}.\beta_2) \in \mathbf{DR} \mid (\exists\ j.\ i<j<k \wedge \beta \stackrel{*}{\Rightarrow} \alpha\mathbf{[i\ j]} \wedge \mathbf{A} \stackrel{+}{\Rightarrow} \alpha\mathbf{[j\ k]})\}$$

$$)$$

This in turn is equal to

$$\mathbf{\{(B} \rightarrow \beta\mathbf{A}\beta'.\beta_3) \in \mathbf{DR} \mid (\exists\ j.\ i<j<k \wedge \beta \stackrel{*}{\Rightarrow} \alpha\mathbf{[i\ j]} \wedge \mathbf{A} \stackrel{+}{\Rightarrow} \alpha\mathbf{[j\ k])} \wedge \beta' \stackrel{*}{\Rightarrow} \#\}$$

This is the set of rules that derive α[i k] using many symbols, as noted above.

Lemma 4.2. Suppose (length α) > 1 and S is the set of dotted rules that derive α using many symbols. The set of symbols that derive α is (close (finished S)).

Proof: By induction as in Lemma 3.2.

Definitions. Let RuleTable' be (SkipEmpty $\{(A \rightarrow .\alpha) \mid (A \rightarrow \alpha) \in P\}$) = $\{(A \rightarrow \alpha.\alpha') \in DR \mid \alpha \Rightarrow \#\}$. If S is a set of symbols let (NewRules' S) be (SkipEmpty (AdvanceDot RuleTable' S)).

Lemma 4.3. If S is the set of symbols that derive α, the set of dotted rules that derive α using one symbol is (NewRules' S).

Proof: Expanding definitions gives

```
(SkipEmpty (AdvanceDot {(A → β.β1) ∈ DR | β ⇒ #)
                       (C | C ⇒ α))
) =

(SkipEmpty ((A → βC.β2) ∈ DR | β ⇒ # ∧ C ⇒ α)
) =

{(A → βCβ'.β3) ∈ DR | β ⇒ # ∧ C ⇒ α ∧ β' ⇒ #}
```

This is the set of dotted rules that derive α using one symbol, by definition.

Let α be a string of length L. For $0 \leq i < k \leq L$, define

```
(dr i k) =
  (let rules1 = (SkipEmpty (∪ i<j<k (AdvanceDot (dr i j)
                                                (∪ (finished (dr j k))
                                                   (if j+1=k (α[j k]} ∅)))))
    (let rules2 = (NewRules' (close (if i+1=k {α[i k]) (finished rules1))))
       (∪ rules1 rules2)
  ))
```

Theorem 4.1 (dr i k) is the set of dotted rules that derive $\alpha[i\ k]$.

Proof: By induction on the length of $\alpha[i\ k]$ as in the proof of theorem 3.1, but with Lemmas 4.1, 4.2 and 4.3 replacing 3.1, 3.2 and 3.3 respectively.

A.5 The Parser with Top-Down Filtering

We have described two parsers that set (dr i k) to the set of dotted rules that derive $\alpha[i\ k]$. We now consider a parser that uses top-down filtering to eliminate some useless rules from (dr i k). Let us say that A follows β if the start symbol derives a string beginning with βA. A dotted rule $(A \rightarrow \chi)$ follows β if A follows β. The new algorithm will set (dr i k) to the set of dotted rules that derive $\alpha[i\ k]$ and follow $\alpha[0\ i]$.

If A derives a string beginning with B we say that A can begin with B. The new algorithm requires a prediction table, which contains all pairs [A B] such that A can begin with B. Let P_1 be the set of pairs [A B] such that $(A \rightarrow \beta B \beta')$ is a rule and $\beta \Rightarrow \#$. Let P_{n+1} be $P_n \cup$ (LinkProd P_n P_1).

Lemma 5.1. The set of pairs [A B] such that A can begin with B is the union of P_n for all $n \geq 1$.

Proof: By induction on the tree by which A derives a string beginning with B. Details are left to the reader.

Our problem is to construct a finite representation for the prediction table. To see why this is difficult, consider a grammar containing the rule

((f (s :x)) → (f :x) A)

Computing LinkProd as in Theorem 1 gives us the following pairs of terms:

```
[ (f (s :x)) (f :x) ]
[ (f (s (s :y))) (f :y) ]
[ (f (s (s (s :z)))) (f :z) ]
. . . . . .
```

Thus if we try to build the prediction table in the obvious way, we get an infinite set of pairs of terms.

The key to this problem is to recognize that it is not neccesary or even useful to predict every possible feature of the next input. It makes sense to predict the presence of traces, but predicting the subcategorization frame of a verb will cost more than it saves. To avoid predicting certain features, we use a weak prediction table—that is, a set of pairs of symbols that properly contains the set of all [A B] such that $A \stackrel{*}{\Rightarrow} B$. This weak prediction table is guaranteed to eliminate no more than what the ideal prediction table eliminates. It may leave some dotted rules in (dr i k) that the ideal prediction table would remove, but it may also cost less to use.

We cannot build the ideal prediction table for every grammar, but we can build a weak prediction table for every grammar. Let Q_1 be a set of terms such that $P_1 \subseteq$ (ground Q_1). Define

$$(\mathbf{LP\ Q\ Q'}) = \{(\mathbf{s\ [x\ z]}) \mid (\exists\ \mathbf{x, y'}.\ \mathbf{[x\ y] \in Q \wedge [y'\ z] \in Q' \wedge s = m.g.u.\ of\ y\ and\ y'})\}$$

By Theorem 2.1, (ground (LP Q Q')) = (LinkProd (ground Q) (ground Q')). Let Q_{i+1} equal $Q_i \cup$ (LP Q_i Q_1). Then by Lemma 2.3,

$$(\cup\ \mathbf{i \geq 1\ P_i}) \subseteq (\mathbf{ground} \cup\ \mathbf{i \geq 1\ Q_i})$$

That is, the union of the Q_i's represents a weak prediction table. Thus we have shown that if a weak prediction table is adequate, we are free to choose any Q_1 that subsumes P_1.

Suppose that Q_D subsumes (LP Q_D Q_1). Then (ground (LP Q_D Q_1)) $\subseteq$ (ground Q_D) and (ground Q_{D+1}) = (ground Q_D). Since (ground Q_{i+1}) is a function of (ground Q_i) for all i, it follows that (ground Q_i) = (ground Q_D) for all i $\geq$ D, so (ground Q_D) = $\cup$ i $\geq$ 1 (ground Q_i). That is, Q_D is a finite representation of a weak prediction table. Our problem is to choose a Q_1 that subsumes P_1 so that Q_D subsumes Q_{D+1} for some D.

Let t_1 and t_2 be types. In Section A.1 we defined $t_1 > t_2$ if there is a function letter of type t_1 that has an argument of type t_2. Let $>^*$ be the transitive closure of $>$; a type t is cyclic if $t >^* t$, and a term is cyclic if its type is cyclic. P_1 is equal to

{[A B] | (A → .Bβ) ∈ RuleTable'}

so we can build a representation for P_1. Let us form Q_1 from this representation by replacing all cyclic terms that are not contained in cyclic terms with distinct new variables. For example, if "cons" and "nil" belong to the cyclic type List, we will turn

(f (cons A nil) (cons B (cons C ...)))

into

(f :x :y)

Then $P_1 \subseteq$ (ground Q_1), and Q_1 contains no function letters of cyclic types. The following lemma shows that this Q_1 allows us to build a finite representation of a prediction table.

Lemma 5.2. Let Q_1 be a set of pairs of terms that contains no function letters of cyclic types, and let Q_i be as defined above for all $i > 1$. Then for some D Q_D subsumes (LP Q_D Q_1).

Proof: Note first that since unification never introduces a function letter that did not occur in the input, Q_i contains no function letters of cyclic type for any $i \geq 1$.

Let C be the number of non-cyclic types in the language. Then the maximum depth of a term that contains no function letters of cyclic types is C+1. For consider a term as a labeled tree, and consider any path from the root of such a tree to one of its leaves. The path can contain at most one variable or function letter of each non-cyclic type, plus one variable of a cyclic type. Then its length is at most C+1.

Consider the set S of all pairs of terms in L that contain no function letters of cyclic types. Let us partition this set into equivalence classes, counting two terms equivalent if they are alphabetic variants. We claim that the number of equivalence classes is finite. Since there is a finite bound on the depths of terms in S, and a finite bound on the number of arguments of a function letter in S, there is a finite bound V on the number of variables in any term of S. Let $v_1 ... v_K$ be a list of variables containing V variables from each type. Then there is a finite number of pairs in S that use only variables from $v_1 ... v_K$; let S' be the set of all such pairs. Now each pair p in S is an alphabetic variant of a pair in S', for we can replace the variables of p one-for-one with variables from $v_1 ... v_K$. Therefore the number of equivalence classes is no more than the number of elements in S'. We call this number E. We claim that Q_D subsumes (LP Q_D Q_1) for some $D \leq E$.

To see this, suppose that Q_i does not subsume (LP Q_i Q_1) for all $i < E$. If Q_i does not subsume (LP Q_i Q_1), then Q_{i+1} intersects more equivalence classes than Q_i does. Since Q_1 intersects at least one equivalence class, Q_E intersects all the equivalence classes. Therefore Q_E subsumes (LP Q_E Q_1), which was to be proved.

This lemma tells us that we can build a weak prediction table for any grammar by throwing away all subterms of cyclic type. In the worst case such a table might be too weak to be useful, but our experience suggests that for natural grammars a prediction table of this kind is very effective in reducing the size of the (dr i k) 's. In the following discussion we will assume that we have a complete prediction table; at the end of this section we will once again consider weak prediction tables.

Definitions. If S is a set of symbols, let (first S) = S $\cup$ { B | ($\exists$ A $\in$ S. [A B] $\in$ PredTable }. If PredTable is indeed a complete prediction table, (first S) is the set of symbols B such that some symbol in S can begin with B. If R is a set of dotted rules let (next R) = {B | ($\exists$ A,β,β'. (A $\rightarrow$ β.Bβ') $\in$ R }.

The following lemma shows that we can find the set of symbols that follow α[0 j] if we have a prediction table and the sets of dotted rules that derive α[i j] for all i < j.

Lemma 5.3. Suppose that for $0 \le i < j$, (S i) is the set of dotted rules that follow α[0 i] and derive α[i j]. Then the set of symbols that follow α[0 j] is

```
(first (if j = 0
           { (start) }
           (∪ 0≤i<j (next (S i,)))))
```

Proof. We show first that every member of the given set follows α[0 j]. If j = 0, certainly every member of (first {(start)}) follows α[0 0] = #. If j > 0, suppose that C follows α[0 i], (C $\rightarrow$ βBβ') is a rule, and $\beta \Rightarrow^* \alpha$[i j]; then clearly B follows α[0 j].

Next we show that if A follows α[0 j], A is in the given set. We prove by induction on d that if (start) $\Rightarrow^*$ α[0 j]Aα' by a tree t, and the leaf corresponding to the occurence of A after α[0 j] is at depth d in t, then A belongs to the given set. If d = 0 then A = (start) and j = 0. We must prove that (start) $\in$ (first [(start)]), which is obvious.

If d > 1 there are two cases. Suppose first that the leaf n corresponding to the occurence of A after α[0 j] has younger brothers dominating a non-empty string. Then the father of n is admitted by a rule of the form (C $\rightarrow$ βAβ'). C is the label of the father of n, and β consists of the labels of the younger brothers of n in order. Then $\beta \Rightarrow^* \alpha$[i j], where $0 \le i < j$. Removing the descendants of n's father from t gives a tree t' whose yield is α[0 i]Cα'. Therefore C follows α[0 i]. We have shown that (C $\rightarrow$ βAβ') is a rule, C follows α[0 i], and $\beta \Rightarrow^* \alpha$[i j]. It follows that A belongs to the set given in the theorem.

Finally suppose that the younger brothers of n dominate the empty string in t. Then if C is the label of n's father, C can begin with A. Removing the descendants of n's father from t gives a tree t' whose yield begins with α[0 j]C. Then C belongs to the given set by induction hypothesis. If C $\in$ (first X) and C can begin with A, then A $\in$ (first X). Therefore A belongs to the given set. This completes the proof.

We are finally ready to present the analogues of lemmas 3.1, 3.2 and 3.3 for the parser with prediction.

Lemma 5.4. Let α be a non-empty string. Suppose that for $0 \le i < k \le$ (length α), (S i j) is the set of dotted rules that follow α[0 i] and derive α[i j], while (S' j k) is the set of symbols that follow α[0 j] and derive α[j k]. The set of dotted rules that follow α[0 i] and derive α[i k] using many symbols is

$$\mathbf{(SkipEmpty} \cup \underset{i<j<k}{\mathbf{(AdvanceDot\ (S\ i\ j)\ (S'\ j\ k))}}$$

Proof: Expanding definitions and using the same argument as in Lemma 3.1 we have

$$\mathbf{(SkipEmpty} \cup \underset{i<j<k}{\mathbf{(AdvanceDot\ \{(B \rightarrow \beta.\beta_1) \in DR \mid B\ follows\ \alpha[0\ i] \wedge \beta \stackrel{*}{\Rightarrow} \alpha[i\ j]\}\ \{A \mid A\ follows\ \alpha[0\ j] \wedge A \stackrel{*}{\Rightarrow} \alpha[j\ k]\}))} =$$

$$\mathbf{(SkipEmpty\ \{(B \rightarrow \beta A.\beta_2) \in DR \mid B\ follows\ \alpha[0\ i] \wedge (\exists\ j.\ i<j<k \wedge \beta \stackrel{*}{\Rightarrow} \alpha[i\ j] \wedge A\ follows\ \alpha[0\ j] \wedge A \stackrel{*}{\Rightarrow} \alpha[j\ k]))}$$

)

If B follows $\alpha[0\ i]$, $(B \rightarrow \beta A \beta_2)$ is a rule, and $\beta \stackrel{*}{\Rightarrow} \alpha[i\ j]$, then A follows $\alpha[0\ j]$. Therefore the statement that A follows $\alpha[0\ j]$ is redundant and can be deleted, giving

$$\mathbf{(SkipEmpty\ \{(B \rightarrow \beta A.\beta_2) \in DR \mid B\ follows\ \alpha[0\ i] \wedge (\exists\ j.\ i<j<k \wedge \beta \stackrel{*}{\Rightarrow} \alpha[i\ j] \wedge A \stackrel{*}{\Rightarrow} \alpha[j\ k])\}}$$

)

This in turn is equal to

$$\mathbf{\{(B \rightarrow \beta A \beta'.\beta_3) \in DR \mid B\ follows\ \alpha[0\ i] \wedge (\exists\ j.\ i<j<k \wedge \beta \stackrel{*}{\Rightarrow} \alpha[i\ j] \wedge A \stackrel{*}{\Rightarrow} \alpha[j\ k]) \wedge \beta' \stackrel{*}{\Rightarrow} \#)}$$

This is the set of dotted rules that follow $\alpha[0\ i]$ and derive $\alpha[i\ k]$ using many symbols.

Lemma 5.5. If (length α) > 1 and S is the set of dotted rules that follow $\alpha[0\ i]$ and derive $\alpha[i\ j]$ using many symbols, then (close (finished S)) contains all symbols that follow $\alpha[0\ i]$ and derive $\alpha[i\ j]$, and every symbol in this set derives $\alpha[i\ j]$.

Proof: It is easy to show that every symbol in the given set derives $\alpha[i\ j]$. Suppose A follows $\alpha[0\ i]$ and derives $\alpha[i\ j]$. Then by Lemma 4.2 there is a dotted rule $(B \rightarrow \beta.)$ such that $\beta \stackrel{*}{\Rightarrow} \alpha[i\ j]$ using many symbols and $A \stackrel{*}{\Rightarrow} B$. Then B follows $\alpha[0\ i]$, so A is in the given the set.

Definition. If S is a set of symbols and R a set of dotted rules, (filter S R) is the set of rules in R whose left sides are in S. In other words, (filter S R) = $\{(A \rightarrow \beta.\beta') \in R. \mid A \in S\}$.

Lemma 5.6. Suppose S contains every symbol that follows $\alpha[0\ i]$ and derives $\alpha[i\ j]$, and every symbol of S derives $\alpha[i\ j]$. Then the set of rules that follow $\alpha[0\ i]$ and derive $\alpha[i\ j]$ using one symbol is (filter { A | A follows $\alpha[0\ i]$ } (NewRules' S)).

Proof: By Lemma 4.3 we know that every dotted rule in the given set derives $\alpha[i\ j]$ using one symbol, and certainly they all follow $\alpha[0\ i]$. Consider any dotted rule that follows $\alpha[0\ i]$ and derives $\alpha[i\ j]$ using one symbol; it can be written in the form $(A \rightarrow \beta E \beta'.\beta_1)$, where β and β' derive # and B derives $\alpha[i\ j]$. Since A follows $\alpha[0\ i]$ and $\beta \stackrel{*}{\Rightarrow} \#$, B follows $\alpha[0\ i]$. Therefore $B \in S$ and the rule is included in the given set.

Let α be a string of length L. For $0 \le i < k \le L$, define

```
(pred j) =
(first (if j = 0
           ((start))
           (∪ 0≤i<j (next (dr i j)))
       ))

(dr i k) =
  (let rules1 = (SkipEmpty (∪ i<j<k (AdvanceDot (dr i j)
                                                (∪ (finished (dr j k))
                                                   (if j+1=k {α[j k]} ∅)))))
    (let rules2 = (filter (pred i)
                          (NewRules' (close (if i+1=k (α[i k]}
                                                      (finished rules1)))))
      (∪ rules1 rules2)
  ))
```

Theorem 5.6 For $0 \le k \le L$, (pred k) is the set of symbols that follow $\alpha[0\ k]$ and for $0 \le i < k$ (dr i k) is the set of dotted rules that follow $\alpha[0\ i]$ and derive $\alpha[i\ k]$.

Proof: We argue by induction on k. The base case is obvious. If $k > 0$, we first show by induction on the length of $\alpha[i\ k]$ that (dr i k) is as claimed for all $i < k$. Consider any j such that $i < j < k$. By the hypothesis of the induction on k, (dr i j) has the desired value. By the hypothesis of the induction on the length of $\alpha[i\ k]$, (dr j k) has the desired value. Then $rules_1$ is the set of dotted rules that follow $\alpha[0\ i]$ and derive $\alpha[i\ k]$ using many symbols, by Lemma 5.3. Next we show that the argument of NewRules' is a set S such that every member of S derives $\alpha[i\ k]$ and every symbol that follows $\alpha[0\ i]$ and derives $\alpha[i\ k]$ is in S. If i+1=k we have (close $\{\alpha[i\ k]\}$) is the set of symbols that derive $\alpha[i\ k]$, which has both properties. If $i+1 < k$ then Lemma 5.4 shows that (close (finished x)) has both properties. (pred i) is the set of symbols that follow $\alpha[0\ i]$ by the hypothesis of the induction on k so by Lemma 5.5, $rules_2$ is the set of dotted rules that follow $\alpha[0\ i]$ and derive $\alpha[i\ k]$ using one symbol. Then the union of $rules_1$ and $rules_2$ is the desired value of (dr i k). This completes the induction on the length of $\alpha[i\ k]$. Since (dr i k) has the desired value for $i < k$, (pred k) is the set of symbols that follow $\alpha[0\ k]$ by Lemma 5.2. This completes the induction on k and ends the proof.

Corollary: (start) $\in$ (finished (dr 0 L)) iff α is a sentence of the language generated by G.

We have proved the correctness of the parser when it uses an ideal induction table. We must still consider what happens when the parser uses a weak prediction table.

Theorem 5.7. If PredTable contains the set of all [A B] such that A can begin with B, then (start) $\in$ (finished (dr 0 L)) iff α is a sentence of the language generated by G.

Proof: Note that the parser with filtering always builds a smaller (dr i k) than the parser without filtering. Since all the operations of the parser are monotonic, this is an easy induction. So if the parser with filtering puts (start) in (dr 0 L) the parser without filtering will do this also, implying that α is a sentence. Note also that the parser with filtering produces a larger (dr i k) given a larger PredTable (again ,this follows easily because all operations in the parser are monotonic). So if α is a sentence the parser with the ideal prediction table includes (start) in (dr 0 L), and so does the parser with the weak prediction table.

A.6 Discussion and Implementation Notes

We have described a parser for a formalism simpler than many formalisms called "unification grammars". There are no meta-rules, no default values of features, no general agreement principles (Gazdar et. al. 1986). We have found this formalism adequate to describe a substantial part of English syntax—at least, substantial by present-day standards. Our grammar currently contains about 400 syntactic rules, not counting simple rules that introduce single terminals. It includes a thorough treatment of verb subcategorization and less thorough treatments of noun and adjective subcategorization. It covers major construction types: raising, control, passive, subject-aux inversion, imperatives, wh-movement (both questions and relative clauses), determiners, and comparatives.

It is clear that some generalizations are being missed. For example, to handle passive we enumerate by hand the rules that other formalisms would derive by meta-rule. We are certainly missing a generalization here, but we have found this crude approach quite practical—our coverage is wide and our grammar is not hard to maintain. Nevertheless we would like to add meta-rules and probably some general feature-passing principles. We hope to treat them as abbreviation mechanisms—we would define the semantics of a general feature-passing principal by showing how a grammar using that principal can be translated into a grammar written in our original formalism. We haven't done this yet, and from the theoretical standpoint that is a weakness of our work.

There is another way to generalize the formalism—by replacing Robinson's unification with a more general matching device. Our approach is well suited to this kind of generalization because we maintain a sharp separation between the details of unification and the parsing mechanism. We proved in part A.2 that unification allows us to compute certain functions and predicates on sets of grammatical expressions—symbolic products, unions, and so forth. In parts A.3 and A.4 we assumed that these functions were available as primitives and used them to build bottom-up parsers. Nothing in parts A.3 and A.4 depends on the details of unification. If we replace standard unification with another mechanism we have only to re-prove the results of part A.2 and the correctness theorems of parts A.3 and A.4 follow at once. To see that this is not a trivial result, notice that we failed to maintain this separation in part A.5. In order to show that one can build a complete prediction table, we had to consider the details of unification: we mentioned terms like "alphabetic variant" and "subsumption". We have presented a theory of bottom-up parsing that is general in the sense that it does not rely on a particular pattern-matching mechanism—it applies to any mechanism for which the results of part A.2 hold. We claim that these results should hold for any reasonable pattern-matching mechanism; the reader must judge this claim by his or her own intuition.

Let us consider some proposed generalizations of Robinson's unification. A current favorite is the so-called disjunction mechanism. This allows one to represent the three terms (f (a)), (f (b)), (f (c)) as a single term: (f (or (a) (b) (c))). It can be formalized as follows. A reduction of a term is defined recursively: a reduction of (or x y) is any reduction of x or y, and if f is not equal to "or", a reduction of (f x y) is any expression of the form (f x' y') where x' and y' are reductions of x and y respectively. A ground instance of a term t is a reduction of a substitution instance of t that contains no variables. Given these definitions one can devise a more general unification algorithm and re-prove the results of part A.2 (the idea is to treat the choice of a value for a disjunction just as Robinson treats

the choice of a substitution for a variable). The correctness theorems for bottom-up parsing follow at once. More general versions of disjunction are possible and probably desirable.

In the author's view, disjunction is the only extension to Robinson's unification that is clearly required in natural language grammars. It may seem plausible to argue that negation is needed, because the base form of an English verb, when read as an indicative, is not third person singular. On the other hand, why not use disjunction to say that the base form is either first person, second person, or third plural? There is no reason to think that any linguistically significant generalization is being missed here.

Our implementation is a Common Lisp program on a Symbolics Lisp Machine. The algorithm as stated is recursive, but the implementation is a chart parser. It builds a matrix called "rules" and sets rules[i k] equal to (dr i k), considering pairs [i k] in the same order used for the induction argument in the proof. It also builds a matrix "symbols" and sets symbols[i k] to the set of symbols that derive α[i k], and a matrix "pred" with pred[i] equal to the set of symbols that follow α[0 i]. Currently the standard parser does not incorporate prediction. We have found that prediction reduces the number of symbols in the matrix "symbols" by a factor of more than 4, but the cost of prediction is so great that a purely bottom-up parser runs faster.

Our program uses a variation of Boyer and Moore's structure-sharing technique. This means that instead of applying a substitution s to a term t, we use the pair [s t] as a representation for the result of applying s to t. The original version allowed one to unify two expressions in this structure-sharing representation. We have found it more efficient to insist that in each unification, at most one term is in the structure-sharing representation. This allows us to represent a substitution as a simple association list, using the function "assoc" to retrieve the substitutions that have been made for variables. To avoid unifying two expressions in the structure-sharing representation, we must occasionally translate an expression from structure-sharing to the standard representation. It suffices to make sure that all the terms appearing in the matrix of symbols and the matrix of predictions are in the standard representation. It is naturally cheaper to do this translation for single terms rather than whole rules.

The other optimizations are fairly obvious. As usual we skip the occur check in our unifications (as long as there are no cyclic types this is guaranteed to be safe). In each symbolic product one set is indexed by the topmost function letter of the term to be matched, which saves a good number of failed unifications. These simple techniques give us tolerable performance—a short sentence is parsed in ten or fifteen seconds.

References.

Barton, G.E., Berwick, R. C., and Ristad, E. S. (1987) *Computational Complexity and Natural Language*. Cambridge: the MIT Press.

Gazdar, G. E., Klein, G., Pullum, G., and Sag, I. (1985) *Generalized Phrase Structure Grammar*. Oxford: Basil Blackwell.

Graham, S., Harrison, M., and Ruzzo, W. (1980) "An Improved Context-free Recognizer." *ACM Transactions on Programming Languages and Systems* 2, 415-462.

Robinson, J. A. (1965) "A Machine-Oriented Logic Based on the Resolution Principle." *Journal of the ACM* 12(1), 23.

Shieber, S. (1985) "Evidence against the Context-freeness of Natural Language." *Linguistics and Philosophy* 8(3), 333.